APPLICATION FOR PATENT

Inventors: Guy Even and Peter-Michael Seidel

Title: PIPELINED MULTIPLICATIVE DIVISION WITH IEEE ROUNDING

The present application claims benefit of U.S. Provisional Patent Application Number 60/421,998 filed October 29, 2002.

FIELD OF THE INVENTION

The present invention relates to numerical data processing apparatus and methods, and, more particularly, to an apparatus and method for performing floating-point division conforming to IEEE formatting and rounding standards.

INCORPORATED MATERIAL

The following material is incorporated for all purposes into the present application in appendices as listed below, following the bibliography:

Appendix A. “Apparatus for pipelined division with IEEE rounding”, by Guy Even and Peter-M. Seidel, U.S. Provisional Patent Application 60/421,998 filed October 29, 2002.

Appendix B. “Pipelined multiplicative division with IEEE rounding”, by Guy Even and Peter-M. Seidel.

Appendix C. “A parametric error analysis of Goldschmidt's division algorithm", by Guy Even, Peter-M. Seidel, and Warren E. Ferguson, in *Proceedings of the 16th IEEE Symposium on Computer Arithmetic*, June 15-18, 2003.

Appendix D. “Deeply pipelined multiplicative division with IEEE rounding using a full size multiplier with redundant feedback”, by Guy Even and Peter-M. Seidel.

BACKGROUND OF THE INVENTION

As the capabilities of microprocessors have increased, hardware modules dedicated to IEEE floating-point division have became common in microprocessors. Appendix A.-Table 1 lists the latencies (i.e., number of cycles required to complete an instruction) and restart times (i.e., number of cycles that elapse until a functional module can engage in a new independent computation) of floating-point division modules in various microprocessors. It is easy to see that floating point division is rather slow compared to addition and multiplication. The relevant IEEE standard is *IEEE Standard for Binary Floating-Point Arithmetic*, ANSI/IEEE 754-1985

When division is performed by numerical iteration using Newton's well-known method, it is necessary to perform two multiplication operations, one of which is dependent on the result of the other, thereby requiring that the multiplication operations be performed sequentially. This requirement limits the speed and efficiency of division operations using Newton's method. In contrast, the well-known algorithm of Goldschmidt, which also requires two multiplication operations, relies on two multiplication operations which are independent of one another, and which can therefore be performed simultaneously to improve the speed and efficiency of the division. In terms of processor architecture, Goldschmidt's division algorithm is more amenable to pipelined and parallel implementations.

Unfortunately, however, there is currently in the prior art no satisfactory measure of the error when employing the Goldschmidt algorithm, and without a good measure of error, the inaccuracies of the intermediate iterative computations accumulate and cause the computed result to drift away from the correct quotient. That is, implementations of Goldschmidt's algorithm are not self-correcting. The lack of a general and simple error analysis of Goldschmidt's division algorithm has deterred many designers from implementing Goldschmidt's algorithm. Thus, most implementations of multiplicative division methods have been based on Newton's method in spite of the longer latency due to dependent multiplications in each iteration.

Those who have utilized Goldschmidt's algorithm have had to keep careful track of accumulated and propagated error terms during intermediate computations. Recent implementations of Goldschmidt's division algorithm still rely on an error analysis that over-estimates the accumulated error. Such over-estimates lead to correct results but require a costly full-precision multiplier circuit that wastes hardware resources and causes unnecessary delay, because the multiplier and the initial lookup table are too large.

The lack of an accurate error estimator therefore discourages the use of Goldschmidt's division algorithm, and prevents an efficient realization of the pipeline advantages of Goldschmidt's algorithm when implemented. This results in increased hardware complexity, power consumption, processing time, and latency for division operations.

There is thus a widely recognized need for, and it would be highly advantageous to have, a compact, accurate, and efficient error estimator for a Goldschmidt division algorithm conforming to IEEE formatting and rounding standards. This goal is met by the present invention.

SUMMARY OF THE INVENTION

The present invention is of an apparatus and method conforming to IEEE formatting and rounding standards, which efficiently and accurately estimates error when using Goldschmidt's algorithm. This allows efficient use of pipelining to increase the speed of floating-point division operations.

In an embodiment of the present invention, multiplication is performed by a Booth recoded multiplier that can be fed by:

(a) a redundant Booth recoded operand and a non-redundant binary operand;

(b) two redundant carry-save operands; or

(c) two non-redundant binary operands.

In another embodiment of the present invention, IEEE rounding is performed by a novel "dewpoint" rounding technique that implements dewpoint rounding with back multiplication. Performing back multiplication with an estimated dewpoint allows the use of a half-size multiplier instead of a full-size multiplier in estimating Goldschmidt algorithm error.

Accordingly, yet another embodiment of the present invention utilizes a half-size multiplier in performing Goldschmidt's algorithm, yielding IEEE-correct rounding while offering the advantages of:

(a) reduced hardware;

(b) improved pipeline organization;

(c) fewer clock cycles;

(d) shorter clock cycles;

(e) reduced latency;

(f) increased throughput; and

(g) lower power consumption.

Therefore, according to the present invention there is provided a method for IEEE-rounding a computed quotient in a processor, the computed quotient corresponding to an exact quotient which equals a dividend divided by a divisor, the method including: (a) determining an error range of the exact quotient; (b) determining a first candidate number and a second candidate number from the error range; (c) associating the first candidate number with a first rounding interval containing numbers that are IEEE-rounded to the first candidate number; (d) associating the second candidate number with a second rounding interval containing numbers that are IEEE-rounded to the second candidate number; (e) computing the dewpoint number, which separates the first rounding interval from the second rounding interval; (f) back-multiplying the dewpoint number by multiplying the dewpoint number by the divisor; and (g) comparing the back-multiplied dewpoint number against the dividend to determine whether the first candidate number represents the IEEE-rounded computed quotient, or whether the second candidate number represents the IEEE-rounded computed quotient.

Furthermore, according to the present invention there is provided a Booth multiplier for computing the product of a first operand and a second operand, including: (a) a first stage operative to preparing the first operand and the second operand for the addition of partial products, and operative to recoding the second operand in Booth radix-8 digits, and operative to generating partial products; (b) a second stage having an adder tree operative to compressing the partial products; and (c) a third stage having an adder operative to compressing the carry-save representation of the product to a binary representation.

DESCRIPTION OF THE PREFERRED EMBODIMENTS

The principles and operation of an apparatus and method according to the present invention may be understood with reference to the accompanying description and the material in the appendices, which disclose the detailed mathematical principles, circuit diagrams, examples, and other information to completely explain implementing the invention.

"Dewpoint" Rounding

The present invention discloses a novel rounding procedure for IEEE floating point division (see Appendix D-Section 4), which is herein referred to as "dewpoint" rounding — so-called because of the analogy to the meteorological temperature below which condensation of moisture occurs, and above which condensation of moisture does not occur. The procedure relies on an error range of the quotient that allows for only two candidate numbers for the final IEEE rounded result. Each candidate number is associated with a rounding interval, which is simply the set of numbers that are IEEE-rounded to the candidate number. The *dewpoint* is defined to be the number separating the two rounding intervals. The rounding decision is obtained by comparing the dewpoint against the exact quotient by applying back-multiplication. This comparison determines which of the candidate numbers is the correct IEEE-rounded result. Appendix D-Section 4 discloses the details of a unified dewpoint rounding procedure for all IEEE rounding modes, thereby eliminating the need for rounding tables.

The novel dewpoint rounding of the present invention represents an improvement over current prior-art approaches for implementing IEEE rounding in division, which first compute a "rounding representative" of the exact quotient (i.e., a number that belongs to the same rounding interval to which the exact quotient belongs), and then round the rounding representative.

An optimized implementation of dewpoint rounding and back multiplication is disclosed in detail in Appendix B-Section 7.6 and in Appendix D-Section 5.3.

To compute the dewpoint, use a rounding injection and a dewpoint displacement constant (as detailed in Appendix D-Section 4.2). The rounding injection is added to the computed quotient, which is then truncated. The dewpoint displacement constant is then added to the truncated computed quotient to obtain the dewpoint.

All the intermediate results mentioned here (the computed quotient, the truncated computed quotient, and the dewpoint) are also represented in redundant representation. This means that each of the additions mentioned above can be computed in constant time (i.e., they do not require a carry-propagate adder with a logarithmic delay). This enables a reduction of the four IEEE rounding modes to a single rounding mode, so that each separate mode need not be dealt with separately.

Furthermore, the test to determine which of the candidate numbers represents the proper rounding of the computed quotient involves evaluating whether the quantity (b*$dewpoint - a$) is zero, positive, or negative. It is noted that the dewpoint is very close to the exact quotient a/b, so that the absolute value of this quantity is very small, and the sign (or zero) of this quantity can be determined by the least-significant few bits. Thus, the hardware used to determine the sign (or zero) can be fed by a small subset of the least-significant bits.

Moreover, in yet another embodiment of the present invention, the back-multiplication is split into two half-size multiplication operations that can be performed in consecutive clock cycles using the same multiplier (refer to cycles 8 and

9 in Appendix B-Table 3). The first part of the back-multiplication is done with an estimated dewpoint (as shown in Appendix B-Section 7.6). The estimated dewpoint is computed from the computed quotient of the previous iteration. An apparatus according to this embodiment of the present invention utilizes a half-size multiplier for the dewpoint back-multiplication, thereby reducing hardware requirements.

Multiplier Hardware Optimization

Addition trees in multipliers are not amenable to pipelining. Short clock cycles are therefore not achievable at reasonable cost if the addition tree has too many rows. Booth radix-8 recoding reduces the number of rows in an addition tree from n to $(n+1)/3$.

Booth radix-8 multipliers are usually implemented using a 3-stage pipeline, as follows:

1. precompute the 3x multiple of the first operand of the multiplier and recode the second operand;
2. an addition tree that computes a carry-save representation of the product; and
3. final carry-propagate addition.

Goldschmidt's algorithm performs only two multiplications per iteration. Hence running Goldschmidt's algorithm on a 3-stage pipeline creates unutilized cycles ("bubbles") in the pipeline. These bubbles increase the latency and reduce the throughput. Certain prior-art processors attempt to utilize these bubbles (and increase throughput) by allowing other multiplication operations to be executed during such bubbles.

The present invention discloses a Booth radix-8 multiplier that allows for both operands to be either in nonredundant representation or carry-save representation.

Booth multipliers with one operand in redundant carry-save representation are known in the prior art, but Booth multipliers with both operands in redundant representation conceptually is a novel feature of the present invention, which reduces the 3-stage pipeline to a 2-stage pipeline for all but the last iteration of the algorithm.

The Booth-8 multiplier design that supports operands in redundant representation is not symmetric, in the sense that the first operand and second operand of the multiplier are processed differently during the first pipeline stage. During the first pipeline stage, operands represented as carry-save numbers are processed as follows:

(a) The first operand is compressed and the 3x multiple thereof is computed.
This requires two adders.

> To reduce hardware requirements, an embodiment of the present invention employs the adder from the third pipeline stage for compressing the first operand. In this embodiment, the compression of the first operand appears in Appendix D-Table 2 as an operation that takes place in the third pipeline stage.

(b) The second operand can be partially compressed from carry-save representation before being fed to the Booth recoder. In Appendix D-"Implementation of the dewpoint computation" a recoding method is detailed.

A method for determining the Booth recoding of the dewpoint correction term is detailed in Appendix B-Section 7. This method is based on a bound on the value of the dewpoint correction term, which determines the most significant digit position *i* involved in the computation (*i* = 24 in Appendix B-Figure 3).

A first Booth recoded operand of the dewpoint correction term is computed modulo 2^{-i}, which has either the value of the dewpoint correction term or the value of the dewpoint correction term plus 2^{-i}. A second Booth recoded operand is computed in the same manner, minus 2^{-i}. Only the most significant Booth recoded digit of the second Booth recoded operand needs to be computed. The other digits are the same as in the first Booth recoded operand.

In parallel with the above computations, a signal is computed that indicates whether the first Booth recoded operand represents the dewpoint correction term plus 2^{-i}. If the signal is a zero, the first Booth recoded operand represents the dewpoint correction constant, and is chosen as the Booth recoded operand. If the signal is a one, the Booth rcoded operand represents the dewpoint correction constant plus 2^{-i} and the second Booth recoded operand is chosen as the Booth recoded operand. For example, $2^{-i} = 2^{-24}$ in Appendix B-Figure 3.

In an embodiment of the present invention, a Booth recoded multiplier can be fed by either non-redundant binary operands or by redundant carry-save operands. When applied to a Booth radix-8 multiplier, this enables reducing the feedback latency to two cycles. The prior art features only Booth multipliers with one operand in redundant carry-save representation or signed-digit representation.

The organization of a Booth multiplier according to this embodiment of the present invention has the following stages, as detailed in Appendix D-Section 5.1.

Stage 1. The two operands of the multiplier are prepared for the addition of the partial products in the second stage. The second operand is recoded in Booth radix-8 digits and the partial products are generated. If the second operand is given in carry-save representation, a partial compressor prepares the second operand for the input of a conventional Booth recoder. The recoding can accept either a binary string or a carry-save encoded digit string.

The first operand is processed as follows: The 3x multiple of the operand is computed using an adder. The first operand can be represented in either binary or redundant carry-save representation. For the case where the first operand is encoded as a carry-save digit string, the computation of the 3x multiple is preceded by a 4:2 adder that computes a carry-save encoding of the 3x multiple. This carry-save encoded digit string is compressed to a binary number by the adder. For the case where the first operand is encoded as a carry-save digit string, the binary representation of the operand is also computed by a binary adder. The binary adder from the third pipeline stage can be used for this purpose if available for carry-save feedback operands. This can save an adder in the first pipeline stage.

Stage 2. In the second stage, the partial products are compressed by an adder tree. In addition to the partial products, an additional row can be dedicated for an additive input.

Stage 3. The third stage contains an adder to compress the carry-save representation of the product to a binary representation. This adder can be shared with the first pipeline stage.

Appendix D-Section 6.1 details how a full size multiplier is used in a floating point divider. Appendix B-Section 6 details how a half-sized multiplier is used.

While the invention has been described with respect to a limited number of embodiments, it will be appreciated that many variations, modifications and other applications of the invention may be made.

Bibliography

[1] R.C. Agarwal, F.G. Gustavson, and M.S. Schmookler. Series approximation methods for divide and square root in the power3 processor. In *Proceedings of the 13th IEEE Symposium on Computer Arithmetic*, volume 14, pages 116–123. IEEE, 1999.

[2] S. F. Anderson, J. G. Earle, R. E. Goldschmidt, and D. M. Powers. The IBM 360/370 model 91: floating-point execution unit. *IBM Journal of Research and Development*, January 1967.

[3] P. Beame, S. Cook, and H. Hoover. Log depth circuits for division and related problems. *SIAM Journal on Computing*, 15:994–1003, 1986.

[4] G.W. Bewick. *Fast Multiplication: Algorithms and Implementation.* PhD thesis, Stanford University, March 1994.

[5] Marius A. Cornea-Hasegan, Roger A. Golliver, and Peter Markstein. Correctness proofs outline for Newton-Raphson based floating-point divide and square root algorithms. In Koren and Kornerup, editors, *Proceedings of the 14th IEEE Symposium on Computer Arithmetic (Adelaide, Australia)*, pages 96–105, Los Alamitos, CA, April 1999. IEEE Computer Society Press.

[6] D. DasSarma and D. W. Matula. Faithful bipartite ROM reciprocal tables. In S. Knowles and W. H. McAllister, editors, *Proc. 12th IEEE Symposium on Computer Arithmetic*, pages 17–28, 1995.

[7] M. Daumas and D.W. Matula. Recoders for partial compression and rounding. Technical Report 97-01, Laboratoire de l'Informatique du Parallélisme, Lyon, France, 1997.

[8] M. Daumas and D.W. Matula. A Booth multiplier accepting both a redundant or a non-redundant input with no additional delay. In *IEEE International Conference on Application-specific Systems, Architectures and Processors*, pages 205–214, 2000.

[9] G. Even, S.M. Mueller, and P.M. Seidel. A Dual Mode IEEE multiplier. In *Proceedings of the 2nd IEEE International Conference on Innovative Systems in Silicon*, pages 282–289. IEEE, 1997.

[10] G. Even and W.J. Paul. On the design of IEEE compliant floating point units. In *Proceedings of the 13th Symposium on Computer Arithmetic*, volume 13, pages 54–63. IEEE, 1997.

[11] G. Even and P.-M. Seidel. A comparison of three rounding algorithms for IEEE floating-point multiplication. *IEEE Transactions on Computers, Special Issue on Computer Arithmetic*, pages 638–650, July 2000.

[12] Guy Even and Peter-M. Seidel. Pipelined multiplicative division with IEEE rounding. In *Proceedings of the 21st International Conference on Computer Design*, October 13-15 2003.

[13] Guy Even, Peter-M. Seidel, and Warren E. Ferguson. A parametric error analysis of Goldschmidt's division algorithm. In *Proceedings of the 16th IEEE Symposium on Computer Arithmetic*, June 15-18 2003. Full version submitted to JCSS.

[14] D. Ferrari. A division method using a parallel multiplier. *IEEE Transactions on Computers*, EC-16:224–226, April 1967.

[15] M. J. Flynn. On division by functional iteration. *IEEE Transactions on Computers*, C-19(8):702–706, August 1970.

[16] R.E. Goldschmidt. Applications of division by convergence. Master's thesis, MIT, June 1964.

[17] IEEE standard for binary floating-point arithmetic. ANSI/IEEE754-1985, New York, 1985.

[18] Cristina Iordache and David W. Matula. On infinitely precise rounding for division, square root, reciprocal and square root reciprocal. In Koren and Kornerup, editors, *Proceedings of the 14th IEEE Symposium on Computer Arithmetic (Adelaide, Australia)*, pages 233–240, Los Amlamitos, CA, April 1999. IEEE Computer Society Press.

[19] H. Kabuo, T. Taniguchi, A. Miyoshi, H. Yamashita, M. Urano, H. Edamatsu, and S. Kuninobu. Accurate rounding scheme for the Newton-Raphson method using redundant binary representation. *IEEE Transactions on Computers*, 43(1):43–51, 1994.

[20] D.E. Knuth. *The Art of Computer Programming*, volume 2. Addison-Wesley, 3nd edition, 1998.

[21] E. V. Krishnamurthy. On optimal iterative schemes for high-speed division. *IEEE Transactions on Computers*, C-19(3):227–231, March 1970.

[22] P. Markstein. *Ia-64 and Elementary Functions : Speed and Precision.* Hewlett-Packard Professional Books. Prentice Hall, 2000.

[23] K. Mehlhorn and F.P. Preparata. Area-time optimal division for $t = \omega((logn)^{1+\epsilon})$. *Information and Computation*, 72(3):270–282, 1987.

[24] P. Montuschi and T. Lang. Boosting very-high radix division with prescaling and selection by rounding. *IEEE Transactions on Computers*, 50(1):13–27, 2001.

[25] Silvia M. Mueller and Wolfgang J. Paul. *Computer Architecture. Complexity and Correctness*. Springer, 2000.

[26] J.M. Muller. *Elementary Functions, Algorithms and Implementation.* Birkhauser, Boston, 1997.

[27] S. F. Oberman and M. J. Flynn. Design issues in division and other floating-point operations. *IEEE Transactions on Computers*, 46(2):154–161, February 1997.

[28] Stuart F. Oberman. Floating-point division and square root algorithms and implementation in the AMD-K7 microprocessor. In Koren and Kornerup, editors, *Proceedings of the 14th IEEE Symposium on Computer Arithmetic (Adelaide, Australia)*, pages 106–115, Los Alamitos, CA, April 1999. IEEE Computer Society Press.

[29] W.J. Paul and P.-M. Seidel. On the Complexity of Booth Recoding. *Proceedings of the 3rd Conference on Real Numbers and Computers(RNC3)*, pages 199–218, 1998.

[30] J.H. Reif and S.R. Tate. Optimal size integer division circuits. *SIAM Journal on Computing*, 19(5):912–924, Oct. 1990.

[31] D.M. Russinoff. A mechanically checked proof of IEEE compliance of a register-transfer-level specification of the amd-K7 floating-point multiplication, division, and square root instructions. *LMS Journal of Computation and Mathematics*, 1:148–200, December 1998.

[32] M.R. Santoro, G. Bewick, and M.A. Horowitz. Rounding algorithms for IEEE multipliers. In *Proceedings 9th Symposium on Computer Arithmetic*, pages 176–183, 1989.

[33] E. M. Schwarz, L. Sigal, and T. McPherson. CMOS floating point unit for the S/390 parallel enterpise server G4. *IBM Journal of Research and Development*, 41(4/5):475–488, July/Sept 1997.

[34] E.M. Schwarz. Rounding for quadratically converging algorithms for division and square root. In *Proceedings of the 29th Asilomar Conference on Signals, Systems and Computers*, volume 29, pages 600–603. IEEE, 1996.

[35] P.-M. Seidel. High-speed redundant reciprocal approximation. *INTEGRATION, the VLSI Journal*, 28:1–12, 1999.

[36] P.-M. Seidel. *On the Design of IEEE Compliant Floating-Point Units and their Quantitative Analysis.* PhD thesis, University of Saarland, Computer Science Department, Germany, 1999.

[37] N. Shankar and V. Ramachandran. Efficient parallel circuits and algorithms for division. *Information Processing Letters*, 29(6):307–313, 1988.

[38] P. Soderquist and M. Leeser. Area and performance tradeoffs in floating-point divide and square-root implementations. *ACM Computing Surveys*, 28(3):518–564, September 1996.

[39] O. Spaniol. *Computer Arithmetic - Logic and Design.* Wiley, 1981.

[40] N. Takagi. Arithmetic unit based on a high speed multiplier with a redundant binary addition tree. In *Advanced Signal Processing Algorithms , Architectures and Impleme ntation II, vol. 1566 of Proceedings of SPIE,* pages 244–251, 1991.

Appendix A

Apparatus for Pipelined Division with IEEE Rounding

(This appendix was part of the provisional patent application 60/421,998)

Abstract

We propose optimized pipelined implementations for Goldschmidt's division algorithm with IEEE rounding based on Booth Radix-8 multiplication. The considered optimizations for the quotient approximation are based on a careful general analysis of tight error bounds for the implementation and are accompanied by the utilization of redundant representations, partial compressions, injection-based rounding and rectangular multipliers for the internal computations. For the implementation of IEEE rounding, we introduce the concept of dewpoint rounding, that allows efficient implementation and reduced requirements for the quotient approximation.

On this basis we propose the implementation of different versions of Goldschmidt's division algorithm with different pipeline depths. None of these implementations requires a full-sized multiplier at any stage of the computations. In this way we reduce latency, cost, and enable increased throughput at a reasonable cost. We suggest a full range of pipelining depth: On one extreme is

a 3-stage pipeline with a restart time that simply equals the latency minus the number of pipeline stages. On the other extreme is a fully pipelined design.

A.1 Introduction and Summary

As the number of transistors in microprocessors increased, a hardware module dedicated to IEEE floating point division became common in microprocessors. Table A.1 lists the latencies (i.e., number of cycles required to complete an instruction) and restart times (i.e., number of cycles that elapse till a functional module can engage in a new independent computation) of floating point division modules in various microprocessors. One can readily see that floating point division is rather slow compared to addition and multiplication. In addition, designs based on multiplicative division contain a full precision multiplier, a rather costly circuit.

This paper presents implementations of Goldschmidt's floating point division algorithm [16] with significantly reduced latency and cost. These implementations suggest a reasonable cost-pipelining tradeoff that allows for a full range of restart times. On one extreme is a three-stage pipeline with a restart time that simply equals the latency minus the number of pipeline stages. On the other extreme is a fully pipelined design (i.e. restart time of one cycle).

Both extremes are quite practical. The largest multiplier used in any pipeline stage is half-sized, and hence the implementations are amenable to short clock periods. The cost of the designs is also reasonable. For example, in IEEE double precision, the fully pipelined design contains five rectangular half-sized multipliers (i.e., size 62 × 30) and two quarter-sized multipliers (i.e., size 62 × 15. The number of partial products associated with these multipliers is roughly 3.6 times the number of partial products associated with a full-sized multiplier (i.e. 56 × 56). On the other hand, the three-stage pipeline design contains only a single half sized multiplier.

We refer the reader to Appendix C and the references for a short review of division algorithms.

Related work. Goldschmidt [16] developed a floating point division algorithm in which each iteration requires two independent multiplications. The algorithm maintains three numbers N_i (which converges to the quotient), D_i (which converges two 1), and $F_i = 2 - D_i$ (called the scaling factor). The algorithm converges quadratically. Loosely speaking, this means that the number of bits needed to represent F_i doubles in each iteration. This leads to the motivation of a fully pipelined design using a sequence of multipliers (one or two per iteration), where the second operand's length doubles in this sequence.

Probably the main hurdle in implementing Goldschmidt's algorithm is analyzing the errors due to imprecise intermediate computations. In Appendix C a general error analysis of a version of Goldschmidt's algorithm is presented. The advantage of the error analysis is demonstrated by analyzing a micro-architecture similar to that in [28] and showing that a smaller multiplier could be used. An error analysis similar to that in Appendix C enables one to realize the idea of a sequence of multipliers (one per iteration), where only the last multiplier is "full-sized". We further carry an idea of Goldschmidt utilizing the fact that the scaling factor tends to 1 so that even the last multiplier is "half-sized".

Techniques. The first step in this paper is an extension of the error analysis in Appendix C. One reason for having to extend the analysis is the usage of redundant representations (e.g. carry-save) for intermediate results. Redundant representation incurs two problems: (i) an increase of relative errors due to truncation (which is easily handled by the analysis in Appendix C), and (ii) inability to determine if an intermediate result is less than (or greater than) 1. However, our main motivation for extending the error analysis is to guarantee that the scaling factor F_i (which tends to 1) never falls below 1. For this purpose we introduce an assumption which we call saturated rounding of F_i. Loosely speaking, the fact that the scaling factor F_i tends to one implies that the most significant half of the binary representation of F_i is either $1.00\ldots$ or $0.11\ldots$ [16]. This observation implies that a half-sized multiplier suffices.

The second step in this paper is a new rounding procedure for IEEE floating point division (see

Sec. A.4). We refer to this rounding procedure as *dewpoint rounding*. The procedure relies on an estimate of the quotient that allows for only two rounded results. A rounding interval corresponds to each candidate rounded result. The point that separates these rounding intervals corresponding to the candidate rounded results is called the *dewpoint*. The rounding decision is obtained by comparing the dewpoint and the exact quotient. This method differs previous approaches that (i) compute a rounding representative of the exact quotient (i.e. a number that belongs to the same rounding interval that the exact quotient belongs to), and (ii) round the representative. We present a unified dewpoint rounding procedure for all rounding modes that avoids rounding tables.

We employ additional techniques, among them: (i) A Booth recoded multiplier that can be fed by either non-redundant binary operands or by redundant carry-save operands [7]. This technique when applied to a Booth radix-8 multiplier enables reducing the feedback latency to two cycles. (ii) Injection based rounding is used to implement directed rounding of intermediate results [9]. (iii) Back-multiplication (i.e comparison between the dewpoint and the exact quotient) requires full precision multiplication. Since we only have a half-sized multiplier, two passes through the multiplier are required. We rearrange the sequence of multiplications so that one of these passes takes place during what is otherwise a bubble in the pipeline. Hence, the latency associated with back multiplication is only one cycle.

Organization. In Section A.2 notation is introduced, division is defined, and Goldschmidt's algorithm is presented. An error analysis of Goldschmidt's algorithm appears in Section A.3. This analysis builds on the analysis that appears in Appendix C. In Section A.4 dewpoint rounding is presented. Section A.5 lists various techniques employed to save cost and delay in a hardware implementation. Section A.6 presents a hardware design, the pipeline organization and scheduling, and evaluates design options.

processor	latency				
	ALU	FP add	FP mult	FP div single	FP div double
ULTRA-Sparc 3	1	4(1)	4(1)	17(15)	20(18)
Pentium 3	1	3(1)	5(2)	17(17)	32(32)
Pentium 4	1	5(1)	7(2)	23(23)	38(38)
Itanium	1	5(1)	5(1)	30+(11)*	40+(13)*
AMD Athlon	1	4(1)	4(1)	16(13)	20(17)
Power3	1	4(1)	4(1)	17(13)	21(17)
Motorola G4	1	5(1)	5(1)	21(21)	35(35)
Alpha 21064	1	4(1)	4(1)	34(34)	63(63)
Alpha 21164	1	4(1)	4(1)	19(19)	31(31)
Alpha 21264/21364	1	4(1)	4(1)	12(9)	15(12)
R8000	1	4(1)	4(1)	14(11)	20(17)
R12000	1	2(1)	2(1)	14(12)	21(19)
Proposed Divider (1/2)	-	-	-	9(6)	11(8)
Proposed Divider (1)	-	-	-	9(3)	11(4)
Proposed Divider (2)	-	-	-	9(1)	11(2)
Proposed Divider (3)	-	-	-	9(1)	11(1)

Table A.1: Latencies(restart-times) of floating-point operations in current commercial microprocessors compared to the proposed division implementations.

A.2 Preliminaries

Notation Let $x_i x_{i+1} \cdots x_j \in \{0,1\}^*$ denote a binary string. We often denote this string also by $x[i:j]$. We also sometimes refer to x_i as $x[i]$. Since we deal with fractions (mostly in the binade $[1,2)$), the weight associated with the bit x_i is 2^{-i}. Namely, a fraction is represented by the binary digit string $x_0.x_1x_2 \cdots x_{p-1}$.

Let $\sigma = \sigma_i \sigma_{i+1} \cdots \sigma_j \in \{0,1,2\}^*$ denote a carry-save encoded digit string (i.e. $\sigma_i \in \{0,1,2\}$). A carry-save encoded digit string $x[i:j]$ is represented by two binary vectors $xc[i:j]$ and $xs[i:j]$. Each carry-save digit $x[\ell]$ satisfies $x[\ell] = xc[\ell] + xs[\ell]$. We refer to xc (resp. xs) as the *carry-string* (resp. *sum–string*) of x.

A carry-save encoded digit string $\sigma[i:j]$ may represent two values: an unsigned value and a signed value (often called a two's complement value). The unsigned value represented by $\sigma[i:j]$ is denoted by $|\sigma[i:j]|$ and equals $|\sigma[i:j]| = \sum_{\ell=i}^{j} \sigma_\ell \cdot 2^{-\ell}$. The two's complement value represented by the carry-save representation $\sigma[i:j]$ is denoted by $|\sigma[i:j]|_{two}$ and equals $|\sigma[i:j]|_{two} = -\sigma_i \cdot 2^i + \sum_{\ell=i+1}^{j} \sigma_\ell \cdot 2^{-\ell}$.

Division Operation We consider the following task which captures the main difficulty in IEEE floating-point division. The dividend and the divisor are represented by p-bit binary strings $A[0:p-1]$ and $B[0:p-1]$, where $|A|, |B| \in [1,2)$. Our goal is to compute $Q[0:p-1]$ that is the binary encoding of the rounded value of the normalized quotient. In other words, (i) Let $A' = A$ if $|A| \geq |B|$, and $A' =$ leftshift(A) if $|A| < |B|$. This way, the quotient $|A'|/|B| \in [1,2)$. (ii) let $q = |A'|/|B|$, (iii) round q (according to the IEEE standard). The binary string $Q[0:p-1]$ should satisfy $|Q| = q$.

The precision determines the value of p as follows. In double precision operation $p = 53$, and in single precision $p = 24$. Four rounding modes are defined in the IEEE standard: round towards zero round towards zero (RZ), round to nearest even (RNE), round towards infinity (RI) and round towards minus infinity(RMI).

Goldschmidt's division algorithm In Appendix C a variation of Goldschmidt's division algorithm based on directed rounding is presented and analyzed. The algorithm is listed below as Algorithm 1. The multiplications involved and their dependencies are depicted in figure A.1.

Directed roundings are used for all intermediate calculations. For example, N'_i is obtained by rounding down the product $N'_{i-1} \cdot F'_{i-1}$. We denote by n_i the relative error incurred when $N'_{i-1} \cdot F'_{i-1}$ is rounded down. Rounding down translates to the assumption that $n_i \geq 0$. Similarly, rounding down is used for computing F'_i and rounding up is used for computing D'_i. Our error analysis requires that: (a) the operands are in the range A', B satisfy $A' \in [B, 2B)$ and $B \in [1,2)$, so that $A'/B \in [1,2)$ (b) all the relative errors incurred by directed rounding be at most $1/4$, and (c) we require that $|e_0| + 3d_0/2 + f_0 < 1/2$.

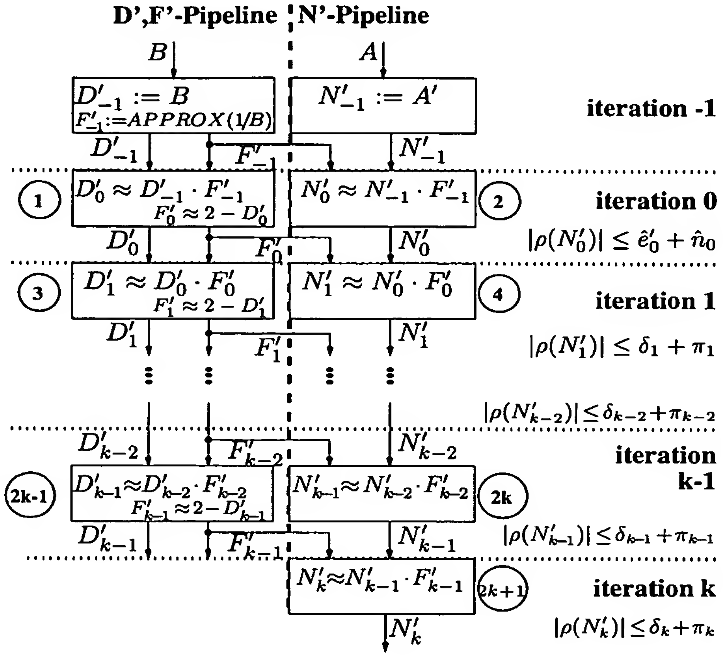
D',F'-Pipeline
N'-Pipeline
B
A
$D'_{-1} := B$
F'_{-1}:=$APPROX(1/B)$
$N'_{-1} := A'$
iteration -1
D'_{-1}
F'_{-1}
N'_{-1}
1
$D'_0 \approx D'_{-1} \cdot F'_{-1}$
$F'_0 \approx 2 - D'_0$
$N'_0 \approx N'_{-1} \cdot F'_{-1}$
2
iteration 0
D'_0
F'_0
N'_0
$|\rho(N'_0)| \leq \hat{e}'_0 + \hat{n}_0$
3
$D'_1 \approx D'_0 \cdot F'_0$
$F'_1 \approx 2 - D'_1$
$N'_1 \approx N'_0 \cdot F'_0$
4
iteration 1
D'_1
F'_1
N'_1
$|\rho(N'_1)| \leq \delta_1 + \pi_1$
$|\rho(N'_{k-2})| \leq \delta_{k-2} + \pi_{k-2}$
D'_{k-2}
F'_{k-2}
N'_{k-2}
iteration k-1
2k-1
$D'_{k-1} \approx D'_{k-2} \cdot F'_{k-2}$
$F'_{k-1} \approx 2 - D'_{k-1}$
$N'_{k-1} \approx N'_{k-2} \cdot F'_{k-2}$
2k
D'_{k-1}
F'_{k-1}
N'_{k-1}
$|\rho(N'_{k-1})| \leq \delta_{k-1} + \pi_{k-1}$
$N'_k \approx N'_{k-1} \cdot F'_{k-1}$
2k+1
iteration k
N'_k
$|\rho(N'_k)| \leq \delta_k + \pi_k$

Figure A.1: Schedule of the Iterations of Goldschmidt's division algorithm using approximate arithmetic and an initial approximation for $1/B$. The numbers in circles indicate the sequence of the multiplications involved. For our implementation a bound on the relative error $\rho(N'_i)$ of iteration i appears in each iteration.

A.3 Extension of error analysis

In this section we present a variation of the error analysis presented in Appendix C. The error analysis presented in Appendix C used a simplifying assumption called *strict directed rounding* (SD-rounding). Satisfying SD-rounding is easy if intermediate results are represented in non-redundant binary representation. In our division algorithm redundant representation is used for intermediate results (i.e. carry-save and borrow-save encoded digit strings). Hence we cannot rely on the SD-rounding assumption.

The analysis uses the same requirements used before, namely: (a) the operands A' and B satisfy $A' \in [B, 2B)$ and $B \in [1, 2)$, (b) all the relative errors incurred by directed rounding are at most $1/4$, and (c) $|e_0| + 3d_0/2 + f_0 < 1/2$.

Algorithm 1 Goldschmidt-Approx-Divide(A', B) - Goldschmidt's division algorithm using approximate arithmetic

```
Require: $0 \le n_i + d_i + f_i \le 1/4$, for all $i$, and $|e_0| + 3d_0/2 + f_0 < 1/2$.
Initialize: $N'_{-1} \leftarrow A'$, $D'_{-1} \leftarrow B$, $F'_{-1} \leftarrow \frac{1-e_0}{B}$.
for $i = 0$ to $k$ do
    $N'_i \leftarrow (1 - n_i) \cdot N'_{i-1} \cdot F'_{i-1}$.
    $D'_i \leftarrow (1 + d_i) \cdot D'_{i-1} \cdot F'_{i-1}$.
    $F'_i \leftarrow (1 - f_i) \cdot (2 - D'_i)$.
end for
Return$(N'_i)$
```

A.3.1 Rounding assumptions

The strict-directed rounding assumption is defined as follows.

Assumption 1 *(***SD-rounding***) Directed rounding up (resp. down) r is* strict *if $x < 1$ implies $r(x) \le 1$ (resp. $x > 1$ implies $r(x) \ge 1$).*

Instead of assuming that SD-rounding holds, we use the following assumptions for iterations $i > 0$:

Assumption 2 *(***saturated rounding***) For $i > 0$,*

$$F'_i = \max\{1, (1 - f_i) \cdot (2 - D'_i)\}.$$

Note that the definition of the relative error f_i in the computation of F'_i is now ambiguous (i.e. suppose $F'_i = 1$ and $(1 - f_i) \cdot (2 - D'_i) < 1$). To avoid ambiguity, define the relative error f'_i as follows:

$$f'_i = \begin{cases} f_i & \text{if } (1 - f_i) \cdot (2 - D'_i) \ge 1 \\ \frac{2-D'_i-1}{2-D'_i} & \text{otherwise.} \end{cases}$$

Note that if $D'_i \le 1$, then $f_i \ge f'_i \ge 0$. This means that saturated rounding reduces the relative error in the computation of F'_i when $D'_i \le 1$.

Assumption 3 *(***precise multiplication by** 1*) Every multiplication by 1 is precise.*

The assumption that multiplication by 1 is always precise means that if $F'_i = 1$, then $D'_{i+1} = D'_i$ and $N'_{i+1} = N'_i$. We say that the algorithm is *stuck* starting with iteration i if $D'_j = D'_i$ and $N'_j = N'_i$, for every $j \geq i$. Note however, that $F'_i = 1$ does not imply that the algorithm is stuck starting with iteration i. The algorithm may not be stuck if $D'_i < 1$ and eventually the relative error f_j is small enough so that $F'_i > 1$. However, if $D'_i \geq 1$, then the algorithm is stuck starting with iteration i.

In the following intermediate values obtained with saturated rounding and precise multiplication by 1 are simply denoted by N'_i, D'_i, and F'_i.

A.3.2 Analysis

With the following theorem we show that even with our current rounding assumptions, N'_i approximating A'/B from below and the relative error $\rho(N'_i) = \frac{A'/B - N'_i}{A'/B}$ can be bounded from above by the same expression as in the error analysis in Appendix C.

Theorem 4 *For every $i > 0$, the relative error $\rho(N'_i) = \frac{A'/B - N'_i}{A'/B}$ satisfies*

$$0 \leq \rho(N'_i) \leq \pi_i + \delta_i, \tag{A.1}$$

where π_i is defined by $\pi_i \triangleq 1 - (1 - n_i) \cdot \prod_{j=0}^{i-1} \frac{1-n_j}{1+d_j} \geq 0$ and where δ_i is defined by

$$\delta_i := \begin{cases} |e_0| + 3d_0/2 & \text{for } i = 0 \\ \delta_{i-1}^2 + f_{i-1} & \text{otherwise} \end{cases}$$

for $i \geq 0$.

Lemma 5 *The following three statements hold:*

a) If $F'_{i-1} = (1 - f_{i-1}) \cdot (2 - D'_{i-1})$ and $D'_{i-1} \in [1 - \delta_{i-1}, 1 + \delta_{i-1}]$, then $D'_i \geq 1 - \delta_i$.

b) If $F'_{i-1} = (1 - f_{i-1}) \cdot (2 - D'_{i-1})$ then $D'_i \leq 1 + d_i$.

c) If $F'_{i-1} = 1 > (1 - f_{i-1}) \cdot (2 - D'_{i-1})$ and $D'_{i-1} \in [1 - \delta_{i-1}, 1]$, then $D'_i \in [1 - \delta_i, 1]$.

Proof: Let $t_j = D'_j - 1$. We first prove Parts a) and b):

$$\begin{aligned} D'_i &= (1 + d_i) \cdot \underbrace{(1 + t_{i-1})}_{D'_{i-1}} \cdot \underbrace{(1 - t_{i-1}) \cdot (1 - f_{i-1})}_{F'_{i-1}} \\ &= \underbrace{(1 + d_i)}_{\geq 1} \cdot \underbrace{(1 - t^2_{i-1})}_{\leq 1} \cdot \underbrace{(1 - f_{i-1})}_{\leq 1} \quad \geq (1 - \delta^2_{i-1} - f_{i-1}) = 1 - \delta_i \quad \textbf{this implies a)} \\ &\leq (1 + d_i) \quad \textbf{this implies b)} \end{aligned}$$

c) Since $F'_{i-1} = 1$, precise multiplication by 1 implies that $D'_i = D'_{i-1}$, so all we need to show is that $D'_i > 1 - \delta_i$. Since $1 > (1 - f_{i-1}) \cdot (2 - D'_{i-1})$, we can write:

$$\begin{aligned} D'_i &> (1 - f_{i-1}) \cdot (2 - D'_{i-1}) \cdot D'_{i-1} \\ &\geq (1 - f_{i-1}) \cdot (1 - \delta^2_{i-1}) \\ &\geq 1 - \delta_i, \end{aligned}$$

and Part c) follows. □

Proof of Theorem 4: In Appendix C Equation (A.1) is proven for iteration i whenever $D'_{i-1} \in [1 - \delta_{i-1}, 1 + \delta_{i-1}]$ and $F'_{i-1} = (1 - f_{i-1}) \cdot (2 - D'_{i-1})$. In fact, these two conditions are used in Appendix C to prove that $1 - \delta_i \leq D'_{i-1} \cdot F'_{i-1} \leq 1$, which implies Equation (A.1) for iteration i. We rely on the proof presented in Appendix C and deal here with the effect of saturated rounding.

Let i_D and i_F be defined as follows:

$$\begin{aligned} i_D &\triangleq \min\{i > 0 : D'_i > 1\} \\ i_F &\triangleq \min\{i > 0 : (1 - f_i) \cdot (2 - D'_i) < 1\} \end{aligned}$$

Note that if $D'_i > 1$, then $(1 - f_i) \cdot (2 - D'_i) < 1$, and hence, $i_F \leq i_D$.

We need to consider the 3 cases: (i.) $i \leq i_F$; (ii.) $i_F < i \leq i_D$; and (iii.) $i > i_D$. Note that case (ii.) only occurs if $i_F \neq i_D$.

Case (i.): One can directly verify that $1 - \delta_0 \leq D'_0 \leq 1 + \delta_0$. Since saturated rounding does not take place in iteration $i = 0$, it follows that $F'_0 = (1 - f_0) \cdot (2 - D'_0)$. Hence the error analysis in Appendix C implies that Equation A.1 holds for $i = 1$.

For $0 < i - 1 < i_F$ we have $D'_{i-1} \leq 1$ and $F'_{i-1} = (1 - f_{i-1}) \cdot (2 - D'_{i-1}) \geq 1$ since $i - 1 < min(i_D, i_F)$. By applying induction and Part a) of Lemma 5, it follows that $D'_{i-1} \in [1 - \delta_{i-1}, 1]$. Hence the proof in Appendix C applies, and Equation A.1 holds for i. It follows that Equation A.1 holds for every $i \in [1, i_F]$.

Case (ii.): By applying induction and Parts a) and c) of Lemma 5, it follows that $D'_{i-1} \in [1 - \delta_{i-1}, 1]$, for every $i - 1 < i_D$. If $F'_{i-1} = (1 - f_{i-1}) \cdot (2 - D'_{i-1})$, then Equation (A.1) holds for i. If $F'_{i-1} = 1 > (1 - f_{i-1}) \cdot (2 - D'_{i-1})$, then by Part c) of Lemma 5, $D'_i \in [1 - \delta_i, 1]$. By precise multiplication by 1 it follows that $D'_i = D'_{i-1}$, and hence $1 - \delta_i \leq D'_{i-1} \cdot F'_{i-1} \leq 1$. Hence the error analysis in Appendix C is applicable, and Equation (A.1) holds for $i \leq i_D$.

Case (iii.) ($i > i_D$)**:** Note that the algorithm is stuck starting from iteration i_D. Since $\pi_{j+1} \geq \pi_j$, it suffices to prove that $0 \leq \rho(N'_{i_D}) \leq \pi_{(i_D+1)}$.

We now prove that $D_{i_D} \in [1, 1 + d_{i_D}]$. The lower bound follows from the definition of i_D. Part b) of Lemma 5 proves that if $F_{i_D-1} = (1 - f_{I_D-1}) \cdot (2 - D_{I_D-1})$, then $D_{i_D} \leq 1 + d_{i_D}$. So all we need to prove is that $F'_{i_D-1} = (1 - f_{i_D-1}) \cdot (2 - D'_{i_D-1})$. Since $D_{i_D} > 1$ it follows that if $D_{i_D-1} \leq 1$, then $F'_{i_D-1} \neq 1$, hence $F'_{i_D-1} = (1 - f_{i_D-1}) \cdot (2 - D'_{i_D-1})$. The minimality of i_D implies that if $D_{i_D-1} > 1$, then $i_D = 1$. But saturated rounding is not used in iteration $i = 0$, so the upper bound on D'_{i_D} holds.

We now expand $\frac{N'_j}{D'_j}$ as follows:

$$\begin{aligned}\frac{N'_j}{D'_j} &= \frac{N'_{j-1}}{D'_{j-1}} \cdot \frac{1-n_j}{1+d_j} \\ &= \frac{A}{B} \cdot \prod_{\ell=0}^{j} \frac{1-n_\ell}{1+d_\ell}.\end{aligned}$$

Thus,

$$N'_j = \frac{A}{B} \cdot \prod_{\ell=0}^{j} \frac{1-n_\ell}{1+d_\ell} \cdot D'_j$$

and

$$\rho(N'_j) = 1 - \prod_{\ell=0}^{j} \frac{1-n_\ell}{1+d_\ell} \cdot D'_j. \tag{A.2}$$

Since $D'_{i_D} \geq 1$, it follows that

$$\begin{aligned}\rho(N'_{i_D}) &\leq 1 - \prod_{\ell=0}^{i_D} \frac{1-n_\ell}{1+d_\ell} \\ &\leq 1 - (1 - n_{(i_D+1)}) \prod_{\ell=0}^{i_D} \frac{1-n_\ell}{1+d_\ell} \\ &= \pi_{(i_D+1)}\end{aligned}$$

which gives the desired upper bound on $\rho(N'_{i_D})$. To prove the lower bound we use $D'_{i_D} \leq 1 + d_{i_D}$ with equation (A.2) as follows:

$$\begin{aligned}\rho(N'_{i_D}) &\geq 1 - (1 + d_{i_D}) \prod_{\ell=0}^{i_D} \frac{1-n_\ell}{1+d_\ell} \\ &= 1 - (1 - n_{i_D}) \prod_{\ell=0}^{(i_D-1)} \frac{1-n_\ell}{1+d_\ell} \\ &= \pi_{i_D} \geq 0.\end{aligned}$$

This completes the proof for the last of the three cases. □

Corollary 6 *For every $i > 0$, $F'_i \in [1, 1 + \delta_i]$*

Proof: If $F'_i = 1$, then the corollary obviously holds. We deal now with the case that $F'_i = (1 - f_i) \cdot (2 - D'_i)$. In the proof of Theorem 4 we showed that $D'_i \geq 1 - \delta_i$, for $i > 0$. Therefore, $F'_i = (1 - f_i) \cdot (2 - D'_i) \leq 1 + \delta_i$. The lower bound follows from Assumption 3 (saturated rounding). □

A.4 IEEE rounding of the quotient

A.4.1 Background on IEEE rounding

The IEEE Floating-Point Standard 754 [17] requires the implementation of all four IEEE rounding modes for all basic arithmetic operations and for all precisions supported. IEEE rounding is supposed to be computed on the exact result of the operation (which is A'/B in our case). For most operations the exact result is not available or too expensive to be computed. It is therefore common practice to compute the IEEE rounding result by computing a rounding representative first and compute IEEE rounding on the representative that leads provably to the same result [10].

In particular for multiplicative division it is even difficult to find a representative quotient from an approximation of the quotient, and it is common practice to involve a back-multiplication by B of the approximation and a comparison with A' for this purpose [19, 34].

A different approach is suggested in [18], which avoids the back-multiplication, but involves computing roughly twice the accuracy of the target precision and making conclusions about rounding from this representation. This approach is based on a proof that shows that, in the binary representation of the exact quotient, the length of runs of zeros (ones) must be limited. We will not follow this approach since it involves one additional iteration for multiplicative division algorithms, adds significant delay to the implementation, and requires very wide intermediate operand representations for the last iteration.

In the sign-magnitude representation of floating-point numbers, the four IEEE rounding modes can be reduced to three rounding modes RZ, RI, and RNU (round to nearest up) based on the numbers' sign [32].

Observation 7 *The exact quotient A'/B cannot be a midpoint between two representable numbers. Therefore RNU and RNE are equivalent rounding modes with respect to division.*

Based on the above reduction and observation we only need to consider the three rounding modes RZ, RI, and RNU.

Injection based rounding [9] was introduced to further simplify rounding. Injection based rounding reduces these three rounding modes to truncation. This reduction is obtained by adding an injection that only depends on the rounding mode.

A.4.2 Directed rounding of carry-save numbers

We propose an implementation of Algorithm 1 in which intermediate results are represented by carry-save encoded digit strings. Since Algorithm 1 uses directed roundings for all intermediate computations, we need to explain how directed rounding is applied to carry-save numbers.

Notation. Consider three binary vectors x, y, z. Let FA denote $3:2$-addition, namely, a line of full-adders fed by x, y, z outputs two binary vectors s, c that satisfy $|x| + |y| + |z| = |s| + |c|$. We regard a carry-save encoded digit string also as two binary vectors. Truncating a carry-save encoded digit string $\sigma[i : j]$ from position q simply means chopping off the tail and leaving only $\sigma[i : q]$. We denote truncation of σ from position q by $\lfloor \sigma \rfloor_q$.

We use injections to define rounding up of a carry-save digit string. The injection creates a carry into position q if the tail $\sigma[q+1 : t]$ is not all zeros. Formally:

Definition 1 *Rounding up of a carry-save digit string $\sigma[0 : t]$ at position q is defined by*

$$RU_q(\sigma[0:t]) \triangleq \lfloor FA(\sigma[0:t], \text{INJ}_{RU}(t,q)) \rfloor_q,$$

where $\text{INJ}_{RU}(t,q) = 2^{-q+1} - 2^{-t}$.

The following lemma provides bounds on the absolute error introduced by truncating and rounding up of a carry-save encoded digit string.

Lemma 8 *1.* $|\sigma[0:t]| - |\lfloor\sigma[0:t]\rfloor_q| \in [0, 2^{-q+1})$.

2. $|RU_q(\sigma[0:t])| - |\sigma[0:t]| \in [0, 2^{-q+1})$.

A.4.3 Dewpoint Rounding

Dewpoints

The concept of dewpoint rounding is inspired by the following concept of a dewpoint in meteorology: the dewpoint is the threshold temperature that separates between the events of rain and no rain.

If the quotient's estimate is close enough to the accurate un-rounded quotient, then rounding only needs to select between two values. We denote by $RC_1 < RC_2$ the two candidate rounded results. Note that $RC_2 = RC_1 + 2^{-p+1}$. For each rounding candidate RC_j, we denote by I_j the rounding interval that comprises the pre-image of RC_j with respect to rounding. Namely, I_j is the set of numbers that are rounded to RC_j. The definition of RZ, RNU, and RI rounding implies that since RC_1 and RC_2 are successive representable numbers, the rounding intervals I_1 and I_2 share an endpoint. This common endpoint is called the *dewpoint*.

To guarantee only two rounding candidates the following assumption on N'_k must hold.

Assumption 9 *Prior to the computation of the dewpoint, the quotient's estimate* N'_k *satisfies:* $0 \leq \rho(N'_k) < 2^{-p}$.

Dewpoint Computation

Given a carry-save encoding $\sigma[0:t]$ of N'_k, we wish to compute the dewpoint and the rounding candidates RC_1 and RC_2. A uniform computation method based on injections is used for all rounding modes.

We use some notation related to the target precision p.

Definition 2 *We refer to multiples of* 2^{-p+1} *as* representable significands *and to odd multiples of* 2^{-p} *as* mid-points.

Since $0 \leq \rho(N'_k) < 2^{-p}$, it follows that $\frac{|A'|}{|B|} \in [N'_k, N'_k + 2^{-p+1})$. A carry-save encoded digit string $\sigma[0:t]$ is used to represent N'_k. Let $tail[p-1:t]$ denote the binary string that satisfies:

$$|tail[p-1:t]| = |\sigma[p:t]|.$$

Truncation of $\sigma[0:t]$ starting at position $p-1$ yields: $|\sigma[0:p-1]| \in (N'_k - 2 \cdot 2^{-p+1}, N'_k]$. By adding in $tail[p-1]$ we obtain a tighter estimate:

$$|\sigma[0:p-1]| + |tail[p-1]| \in (N'_k - 2^{-p+1}, N'_k].$$

Let $RAW \triangleq |\sigma[0:p-1]| + |tail[p-1]|$. We conclude that

$$\frac{|A'|}{|B|} \in [RAW, RAW + 2 \cdot 2^{-p+1}). \tag{A.3}$$

Observe that RAW and $RAW + 2^{-p+1}$ are the only representable significands in this interval. This does not guarantee yet that these are the only two rounding candidates. In RI and RNU the interval $[RAW, RAW + 2 \cdot 2^{-p+1})$ intersects three rounding intervals. We would like this interval to contain at most two rounding intervals.

Rounding mode RZ. In the RZ rounding mode the interval $[RAW, RAW + 2 \cdot 2^{-p+1})$ is the union of exactly two rounding intervals. Hence, the dewpoint in this case is $RAW + 2^{-p+1}$ and the rounding candidates are simply $RC_1 = RAW$ and $RC_2 = RAW + 2^{-p+1}$. This case is depicted in the top part of Fig. A.2.

Other rounding modes. We reduce the rounding modes RI and RNU to RZ by using injections [11] as follows.

Let $INJ_{rnd}(mode)$ denote an injection constant that depends only on the rounding mode. We define $INJ_{rnd}(mode)$ as follows.

$$INJ_{rnd}(mode) := \begin{cases} 0 & \text{for } mode = RZ \\ 2^{-p} & \text{for } mode = RNU \\ 2^{-p+1} - ulp & \text{for } mode = RI. \end{cases}$$

The addition of $INJ_{rnd}(mode)$ reduces RI and RNU to RZ in the following sense [11]:

$$RNU(x) = RZ(x + INJ_{rnd}(RNU))$$
$$RI(x) = RZ(x + INJ_{rnd}(RI)).$$

We would like to compute RC_1, RC_2 and the dewpoint in the cases of RNU and RI by adding $INJ_{rnd}(mode)$. One should observe that $INJ_{rnd}(mode) + |\sigma[p:t]|$ may be greater than or equal to $2 \cdot 2^{-p+1}$. This implies that an extra bit should be used for representing the carry part of the tail. Namely, Let $tail[p-2:t]$ denote the binary string that satisfies:

$$|tail[p-2:t]| = |\sigma[p:t]| + INJ_{rnd}(mode).$$

Define RAW' as follows:

$$RAW' \triangleq |\sigma[0:p-1]| + |tail[p-2:p-1]|.$$

Claim 10 *For every mode* $\in$ *{RZ,RNU,RI}*

$$RAW' - INJ_{rnd}(mode) \leq \frac{|A'|}{|B|} < RAW' - INJ_{rnd}(mode) + 2^{-p+2}.$$

Proof: Recall that $|\sigma[0:t]| \leq \frac{|A'|}{|B|} \leq |\sigma[0:t]| + 2^{-p+1}$. By the definition of $tail[p-2:t]$, it follows that $|\sigma[0:p-1]| + INJ_{rnd}(mode) - 2^{-p+1} < RAW' \leq |\sigma[0:p-1]| + INJ_{rnd}(mode)$. The claim now follows. □

The proof of the following claim can be derived from Fig. A.2.

Claim 11 *For every rounding mode,* $RC_1 = RAW'$ *and* $RC_2 = RAW' + 2^{-p+1}$.

As illustrated in Fig. A.2 the difference between RC_1 and the dewpoint depends only on the rounding mode. We introduce a second injection constant $INJ_{dew}(mode)$ defined as follows:

$$INJ_{dew}(mode) := \begin{cases} 2^{-p+1} & \text{for } mode = RZ \\ 2^{-p} & \text{for } mode = RNU \\ 0 & \text{for } mode = RI, \end{cases}$$

From Fig. A.2 it can be seen that $RC_1 - dewpoint = INJ_{dew}(mode)$, for every rounding mode. Hence the dewpoint is simply computed by adding $RAW' + INJ_{dew}(mode)$.

Dewpoint back multiplication

The rounded result Q is determined as follows:

$$Q = \begin{cases} RC1 & \text{if } \frac{|A'|}{|B|} < dewpoint \\ RC2 & \text{if } \frac{|A'|}{|B|} > dewpoint \\ RC1 & \text{if } \frac{|A'|}{|B|} = dewpoint \text{ and } mode = \text{RI} \\ RC2 & \text{if } \frac{|A'|}{|B|} = dewpoint \text{ and } mode = \text{RZ} \end{cases} \tag{A.4}$$

The case $\frac{|A'|}{|B|} = dewpoint$ and $mode =$ RNU is missing since by Observation 7 it cannot occur.

The comparison $<, =, >$ between $\frac{|A'|}{|B|}$ and *dewpoint* is done by computing the sign of $\mathit{dewpoint} \cdot |B| - |A'|$ and testing if it is zero. This task is often referred to as *back multiplication.*

A.5 Hardware Optimizations

A.5.1 Redundant Feedback & Partial Compression

Our FP-DIV micro-architecture is based on a Booth Radix-8 multiplier. Such multipliers are popular in current microprocessor designs that use short clock periods. Booth Radix-8 multipliers are usually implemented using a 3-stage pipeline (i.e. prepare $3A$, addition tree, and final $2:1$-addition). Goldschmidt's algorithm performs only 2 multiplications per iteration which creates un-utilized cycles (i.e. "bubbles"). The Athlon (in hardware) and Itanium (in software) processors allow other multiplications to be executed during the bubbles that occur during a division operation.

Instead we follow Daumas & Matula [7] and allow redundant operands so that the effective latency is reduced to 2 cycles. This is obtained by feeding back results represented in carry-save format. This design is not symmetric in the sense that the multiplier and the multiplicand are processed differently. The multiplier is partially compressed before being fed to the Booth encoder. Non-redundant representations of the multiplicand and its multiple by 3 are computed by non-redundant adders.

A.5.2 Implementation of injection-based Rounding

Injection values are fed as constants to muxes that select the injection according to the cycle and the rounding mode. The use of injection based rounding reduces directed rounding to truncation. Extra rows in the adder tree are used for adding in injections, feedback from a upper part of a long multiplication, or the complement of A' (during back-multiplication).

A.5.3 Normalization by checking $(A \geq B)$

Suppose that the operands satisfy $|A|, |B| \in [1, 2)$. By comparing them, and shifting A to the left by one position if $|A| < |B|$, we can normalize the quotient. Namely, we can guarantee that the quotient is in the binade $[1, 2)$. This normalization also slightly improves the bound on the error.

A.5.4 Implementation of Saturated Rounding

The definition of saturated rounding from Assumption 2 involves modifications only for the computation of F_i', for $i > 0$.

We propose the following implementation. During each multiplication by F_i' that takes place in stage 2 of the pipeline, check if $F' \leq 1$. If $F_i' \leq 1$, then simply de-activate the clock enable signal of the register that is supposed to store the result of the multiplication. The effect of ignoring the multiplication means that the multiplicand is not changed, i.e., precise multiplication by 1 is obtained.

Note that testing if $F_i' \leq 1$ can be done by testing whether the truncation of D_i' is greater than or equal to 1. Recall that D_i' is represented in carry-save format. To subtract 1 from the truncation of D_i' we simply use a $3:2$-adder (PPM-add) to obtain a borrow-save number. The borrow-save number is fed to a subtracter, and the decision is made according to the value of the sign bit.

A.5.5 Half-size multiplication in the last iteration

The precision in the last iteration must be at least p to obtain a sufficiently close approximation of the quotient. This presumably implies that the length of the second operand (i.e. the multiplier) should be roughly p. Based on the error analysis and, in particular, based on Corollary 6, we are able to implement the last iteration with a single pass through a multiplier, the length of its second operand being roughly $p/2$.

Consider double precision. The last iteration requires computing the product $N_2' = N_1' \cdot F_1'$. Recall that the length of F_1' is roughly $p = 53$, otherwise the relative error f_1' would be too large.

How can we avoid having to multiply N_1' by the full length of F_1'?

Rewrite N_2' as follows:

$$\begin{aligned} N_2' &= N_1' \cdot (1 + (F_1' - 1)) && \text{(A.5)} \\ &= N_1' + N_1' \cdot (F_1' - 1). && \text{(A.6)} \end{aligned}$$

By Corollary 6, $(F_1' - 1) \in [0, \delta_1]$. Hence, we may ignore bit positions to the left of position $j = \lfloor \log_2 \frac{1}{\delta_1} \rfloor$. We conclude that $N_2' = N_1' + N_1' \cdot F_1'[j : t]$, where t is roughly p.

The addition of N_1' to the product $N_1' \cdot F_1'[j : t]$ can be done by using one or two of the extra rows in the adder tree used also for feeding injections. Note that RZ is used for the computation of N_2', hence these extra rows are free during this computation.

A.5.6 Optimized implementation of Dewpoint Rounding and Back multiplication

Recall that selecting the correct rounding candidate is based on the sign-bit (positive, negative, zero) of $\mathit{dewpoint} \cdot |B| - |A'|$.

For the consideration of negative numbers and signs, we need to consider two's complement representations. In particular we consider a representation called two's complement carry-save representation [7]. In this representation the most significand position has a negative weight. Our goal in the back multiplication is to compute the sign of $\alpha = B \cdot \mathit{dewpoint} - A'$, we also want to know if α is precisely zero to determine the condition for rounding mode RI. Let z-sign(α) be two bits, the first signifies if α is negative, and the second bit signifies if α is zero. Hence our goal is to compute z-sign(α).

We employ the multiplier for this purpose using two passes. In the first pass we compute

$$\alpha^H[0:81] \leftarrow B[0:52] \cdot \mathit{dewpoint}[0:29] + (4 - A')[-1:52].$$

In the second pass we compute

$$\alpha^L[29:105] \leftarrow B[0:52] \cdot \mathit{dewpoint}[30:105] + \alpha^H[30:81].$$

Hence, $\alpha = -\alpha^H[0] + \alpha^H[1:29] + \alpha^L[29:105]$. Observe that α is represented by a two's complement carry-save number and that with the help of a half-adder line the carry-save representations of $\alpha^H[1:29]$ and $\alpha^L[29:83]$ could easily be concatenated to a carry save representation of α. It seems that our computation should also deal with digits in positions $[-2:-1]$, instead of considering position $[0]$ to be the most significand position. We explain now why, in fact, it suffices to deal only with digits in positions $[51:105]$.

Consider a binary representation $a_{-2}, a_{-1}, \ldots, a_{105}$ of α, namely:

$$\alpha = -a_{-2} \cdot 2^2 + \sum_{i=-1}^{105} a_i \cdot 2^{-i}.$$

Since $-2^{-52} < \alpha < 2^{-52}$, it follows that $-a_{-2} \cdot 2^2 + \sum_{i=-1}^{51} a_i \cdot 2^{-i} \in \{-2^{-51}, 0\}$. Hence, the bits in positions $[-2, 51]$ are either all zeros or all ones. We conclude that

$$\alpha = 0 \quad \text{iff} \quad \bigvee_{i=52}^{105} a_i = 0, \tag{A.7}$$

$$\alpha < 0 \quad \text{iff} \quad a_{51} = 1. \tag{A.8}$$

A.6 Implementation & Evaluation

This section describes the microarchitectures of the proposed division implementations. We begin with a discussion of the three basic stages that are involved in the computation, followed by a discussion of the scheduling and the required operand widths for IEEE single precision and double precision operands. ¿From the basic three stage microarchitecture we then derive several instatiations, that allow to trade-off hardware cost and throughput.

A.6.1 Basic Microarchitecture

A block diagram with the three stages of our basic microarchitecture is depicted in Fig. A.4. It arranges the blocks and techniques that were introduced in the previous sections around the core of a Radix-8 recoded Multiplier implementation. In the multiplier previously computed products can be fed back to the multiplier either as a multiplicand or as a multiplier. Latency is reduced to by allowing the feed back product to be a carry-save encoded digit string. The multiplier also supports a multiply-add operation (i.e. $A' \times B + C$). The addend (i.e. the number to be added to the product) can be input as a carry-save number or as a binary number, so 2 rows of the adder tree are allocated for the addend.

The multiplication circuitry is divided in to 3 pipeline stages. Below we describe some details of the stages:

1. In the first stage, the multiplicand and multiplier are prepared for the addition of the partial products in the second stage. The multiplier is recoded in Booth Radix-8 digits and the partial products are generated. Following [7], the recoding can accept either a binary string or a carry-save encoded digit string. For the last iteration and IEEE rounding the multiplier is pre-processed by a fixed shift and a redundant dewpoint computation.

   The multiplicand is processed as follows. The $3\times$ multiple of the multiplicand is computed using an adder. A feed back product, encoded as a carry-save digit string, can be used as a multiplicand as follows. The computation of the $3\times$ multiple is preceded by a $4 : 2$-adder that computes a carry-save encoding of the $3\times$ product. This carry-save encoded digit string is compressed to a binary number by the adder. Meanwhile, the binary representation of the multiplicand is computed by the adder in the third pipeline stage. This saves an adder in the first pipeline stage (observe that the adder in the third pipeline is indeed free).

2. In the second stage, the partial products are compressed by a 'half-sized' adder tree. In addition to the partial products, there are two rows that are dedicated for (a) a carry-save number that is fed back from the previous multiplication, or (b) an injection (The injection

is used mainly to reduce a ceiling rounding to a floor rounding.). The partial products and the 2 extra rows are added to produce a carry-save representation of their sum. Before that the inputs for the two extra row are generated or selected.

3. In the third stage, an adder compresses the carry-save number into a binary number. This adder can also be forced just to compute an increment. This operation mode is required to compute RC_2 from RC_1 in the last cycle of the division for rounding. Additional circuitry is installed to check for the condition $F_i' \leq 1$ from the carry-save representation of F_i'. and to check for the condition $\mathit{dewpoint} \cdot B - A < 1$ for the rounding decision from the carry-save representation of this expression. Few additional logic controls the selection between RC_2 from RC_1 based on the evaluation of the condition $\mathit{dewpoint} \cdot B - A < 1$.

A.6.2 Scheduling of the Stages

In this section we present the steps of the division algorithm in detail. The division algorithm for single precision is listed in Table A.2 implementing one iteration of Goldschmidt's algorithm and the division algorithm for double precision is listed in Table A.3 implementing two iterations of Goldschmidt's algorithm. The three columns of the tables correspond to the use of the three pipeline stages of the basic microarchitecture (although there might be more than one instance of a particular stage of the microarchitecture for the division implementation). As usual, we ignore the issues of computing the exponent and sign bit of the quotient, as these are rather straightforward.

A.6.3 Precisions of the Operands

There are several parameters of the implementation that need to be determined, such as the operand widths of the multiplier and the initial approximation. We are targeting the implementaiton for single precision (with $p = 24$) and double precision (with $p = 53$). Because the requirements for single precision are weaker than the requirements for double precision, we are only focusing on double precision in the following.

For the initial reciprocal approcimation there are numerous implementation choices and one could even trade off initial approximation quality against internal operand precision requirements as it is demonstrated in Appendix C. We just assume to use the initial approximation implementation as mentioned in [28]. This gives us $|e0| \leq 2^{-13.75}$. We assume that the initial reciprocal estimate has a representation using less than 16 bits.

To fulfill the requirements for dewpoint rounding according to Assumption 9, we require that $0 \leq \rho(N_2') < 2^{-53}$.

With this background we are ready to determine the operand widths for the rectangular multiplier implementation in our microarchitecture. Note that the multicand input A is used to feed in the values of N_i' and D_i' and that the multiplier input B is used to feed in the values of F_i'.

Operand Width for the Multiplier B In double precision the requirement for the widest operands of the multiplier B are imposed by the representation of the operands F_0' and $(1 - F_1')$. Because operand F_0' is used in carry-save representation, it requires a representation in bit positions $[0 : \lceil -log_2(f_0) \rceil + 1]$. For the operand $(1 - F_1')$ we know from Corollary 6 that $(1 - F_1') \in [0, \delta_1)$. (The case that F_1' does not need to be considered because of our implementation as described in section A.5.4). Because $(1 - F_1')$ is available in binary (see schedule in table A.2), it requires representation in bit positions $[\lfloor -log_2(\delta_1) \rfloor : \lceil -log_2(f_1) \rceil]$.

For a given target approximation there are several different combinations of operand widths for the two operand representations that meet the requirements. Our objective in the search for the optimized combimation depends on the organization of our division implementation and there is a different objective depending on whether the two operands are succesively fed into the same multiplier or into two different multipliers that allow for different bit widths:

Optimization objective 1 (minimum max bit width):

$$minimum_{\text{feasible operand width pairs}} (max\,(\lceil -log_2(f_0) \rceil + 2, (\lceil -log_2(f_1) \rceil - \lfloor -log_2(\delta_1) \rfloor))$$

Optimization objective 2 (minimum accumulated bit positions):

$$\lceil -log_2(f_0) \rceil + 2 + 1 + (\lceil -log_2(f_1) \rceil - \lfloor -log_2(\delta_1) \rfloor)$$

We are only sketching the optimization for objective 1 here. Figure A.3 depicts the feasible bit width combinations and highlights the optimized choice regarding optimization objective 1. In the optimized setting the multiplier is required to be represented with 30 bits. The feasible pairs have been computed with the help of Theorem 4 and the target of $0 \leq \rho(N_2') < 2^{-53}$ while assuming that π_2 only has a minor contribution.

Operand Width for the Multiplicand A For the selected operand bit width for the multplicand, we need to additionally consider the requirements for π_2 with this setting. With Theorem 4 we get the requirement $\pi_2 \leq 2^{-58.53}$ which implies $n_0 \leq 2^{-60.85}$ making it necessary to have the multiplicand represented by 62 bits.

In this way we determine our rectangular multiplier to have the dimensions 62 × 30 bits.

A.6.4 Other Implementation Options & Evaluation

In the previous sections the basic architecture of our division implementation has been described, also how the computation steps need to be scheduled to target IEEE single precision and double precision and how to derive the bit widths for the multiplications.

Based on this general organization, we are proposing several different variations for the realization, that differ from each other by how many hardware components are utilized.

Proposed Design (1/2) With a single instance of the microarchitecture from figure A.4 implementing a three stage pipeline, one can easily check from the scheduling in tables A.2 and A.3 that the latency for single precision is 9 clock cycles (including 1 cycle for reciprocal approximation) and that the latency for double precision is 11 clock cycles. This latency will also be required by the other realizations that we suggest for the implementation. With one 'half-sized' rectangular 62 × 30 bit multiplier one can schedule a new independent division every 6 cycles in single precision and every 8 cycles in double precision. These values of the latencies and restart times are listed in table A.1 to be compared with the corresponding values from the implementations in current microprocessor designs. Note that for both, the latency and the startup time, our proposed design (1/2) shows a significant improvement, while it allows for very fast pipeline stages. Because we only use a 'half-sized' 62 × 30 bit multiplier for this realization, the cost will only be roughly half of the cost of a regular multiplicative division implementation. This justifies the name of Proposed Design (1/2).

Proposed Design (1), (2), and (3) By utilizing more than just one instantiation of the microarchitecture from A.4, it is possible to increase the throughput of the division implementation while increasing the cost of the implementation. The propsed design (1) combines two 62 × 30 bit instances of the three stage pipeline to improve the restart times for single and double precision to 3 and 4 cycles respectively. The propsed design (2) combines three 62 × 30 bit instances and one 62 × 15 bit instance of the microarchitecture, and the propsed design (2) combines five 62 × 30 bit instances and two 62 × 15 bit instance of the microarchitecture. A rough cost estimation classifies these realizations to require about 1×, 2×, and 3× the hardware cost of a conventional multiplicative division implementation, which is the reason why they are named by Proposed Design (1), Proposed Design (2) and Proposed Design (3). The corresponding latencies and restart times for the our designs are listed in Table A.1. In this way we provide a selection of options to allow for trade-offs between hardware cost and throughput of division implementations. The options range from roughly one half up to three times the cost of a regular multiplicative division implementation, and the throughput for double precision ranges from one division every 8 cycles up to the fully pipelined implementation that allows to start a new division every cycle.

cycle	op 1st stage	op 2nd stage	op 3rd stage

1	$F'_{-1}[0:14] \leftarrow \frac{1-e_0}{B}$
	$D'_{-1}[0:61] \leftarrow B$
	recode(F'_{-1})
	prepare($3D'_{-1}$)		

2	$N'_{-1}[0:61] \leftarrow A$ prepare($3N'_{-1}$)	$D'_0[0:61] \leftarrow mul_{RI}(D'_{-1}[0:61], F'_{-1}[0:14])$	
3		$N'_0[0:61] \leftarrow mul_{RZ}(N'_{-1}[0:61], F'_{-1}[0:14])$	compress(D'_0)

4	$F'_0[0:29] \leftarrow 2 - D'_0[0:29]$
	recode($F'_0[0:29]$)
	prepare($3N'_0$)
			compress(N'_0)

5		${N'_1}^H[0:62] \leftarrow$ $mul_{RZ}(N'_0[0:61], F'_0[0:29])$	

6	take *dewpoint*[0 : 29]
	from 3rd stage
	recode(*dewpoint*[0 : 29])
	prepare($3 \cdot B$)
			If $A < B$
			Then $N'_1[0:61] \leftarrow shift\_left({N'_1}^H[1:62])$
			Else $N'_1[0:61] \leftarrow {N'_1}^H[0:61]$).
			compute *dewpoint*[0 : 29]
			through dewpoint injection
			compute rounding candidate RC_1
			through rounding injection.

7		$\alpha[0:90] \leftarrow \quad B[0:61] \cdot dewpoint[0:29] + (4-A)[-1:61]$	
8			compute rounding candidate RC_2 through increment test($\alpha[0:90] < 0$?) perform rounding selection between RC_1 and RC_2.

Table A.2: Schedule of operations of our FP division implementation with IEEE rounding for single precision operands

cycle	op 1st stage	op 2nd stage	op 3rd stage

1	$F'_{-1}[0:14] \leftarrow \frac{1-e_0}{B}$
	$D'_{-1}[0:61] \leftarrow B$
	recode(F'_{-1})
	prepare($3D'_{-1}$)		

2	$N'_{-1}[0:61] \leftarrow A$ prepare($3N'_{-1}$)	$D'_0[0:61] \leftarrow mul_{RI}(D'_{-1}[0:61], F'_{-1}[0:14])$	

3	$F'_0[0:29] \leftarrow 2-(D'_0[0:29]+2\cdot 2^{-29})$
	recode(F'_0)
	prepare($3D'_0$)
		$N'_0[0:61] \leftarrow mul_{RZ}(N'_{-1}[0:61], F'_{-1}[0:14])$	compress(D'_0)

4	prepare($3N'_0$)	$D'_1[0:61] \leftarrow mul_{RI}(D'_0[0:61], F'_0[0:29])$	compress(N'_0)
5		$N'_1[0:61] \leftarrow mul_{RZ}(N'_0[0:61], F'_0[0:29])$	compress(D'_1)

6	$F'_1[26:55] \leftarrow 2^{-25} - D'_1[26:55]$
	recode($F'_1[26:55]$)
	prepare($3N'_1$)
			compress(N'_1)

7	$dewpoint^h[0:29] \leftarrow N'_1[0:29]$
	recode($dewpoint^h[0:29]$)
	prepare($3\cdot B$)
		$N'_2[0:62] \leftarrow$
		$mul_{RZ}(N'_1[0:61], F'_1[26:55]) + N'_1[0:61]$	

8	take $dewpoint_l[30:53]$
	from 3rd stage
	recode($dewpoint_l[30:53]$)
	prepare($3\cdot B$)
		$\alpha^H[0:81] \leftarrow \quad B[0:52]\cdot dewpoint[0:29]$
		$+(4-A')[-1:52]$
			If $A < B$
			Then $N_2'^s[0:61] \leftarrow shift\_left(N_2'^H[1:62])$
			Else $N_2'^s[0:61] \leftarrow N_2'^H[0:61]$).
			compute $dewpoint[0:53]$
			through dewpoint injection
			compute rounding candidate RC_1
			through rounding injection.

9		$\alpha^L[29:105] \leftarrow \quad B[0:52]\cdot dewpoint_l[30:53]$ $+\alpha^H[30:81]$	
10			compute rounding candidate RC_2 through increment test($\alpha[51:105] < 0$?) perform rounding selection between RC_1 and RC_2.

Table A.3: Schedule of operations of the proposed FP division implementation with IEEE rounding for double precision operands

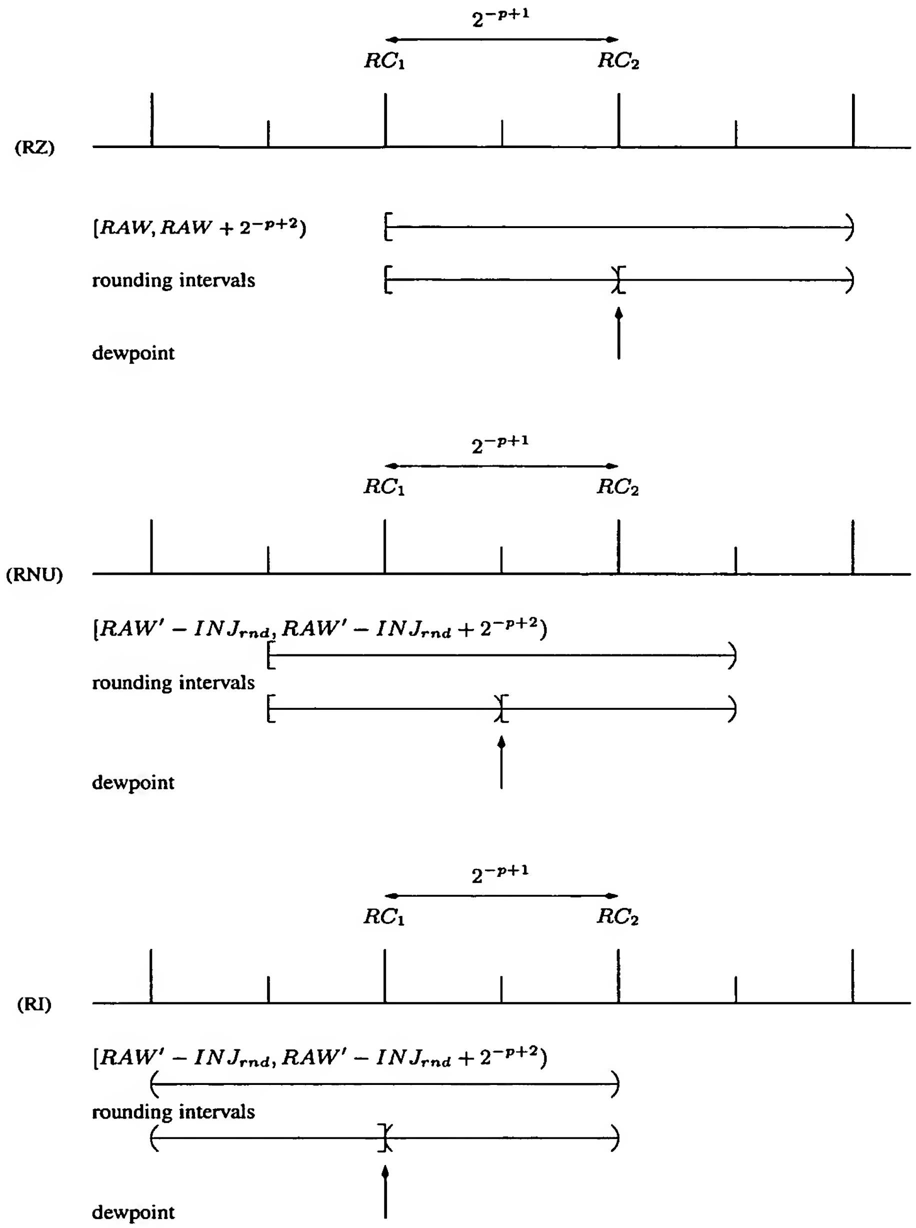

Figure A.2: Computing RC_1, RC_2 and the dewpoint for the three rounding modes RZ, RNU and RI. The injection constant INJ_{rnd} shifts RAW' so that (i) the interval $[RAW' - INJ_{rnd}, RAW' - INJ_{rnd} + 2^{-p+2})$ contains the exact quotient, and (ii) this interval contains exactly two rounding intervals.

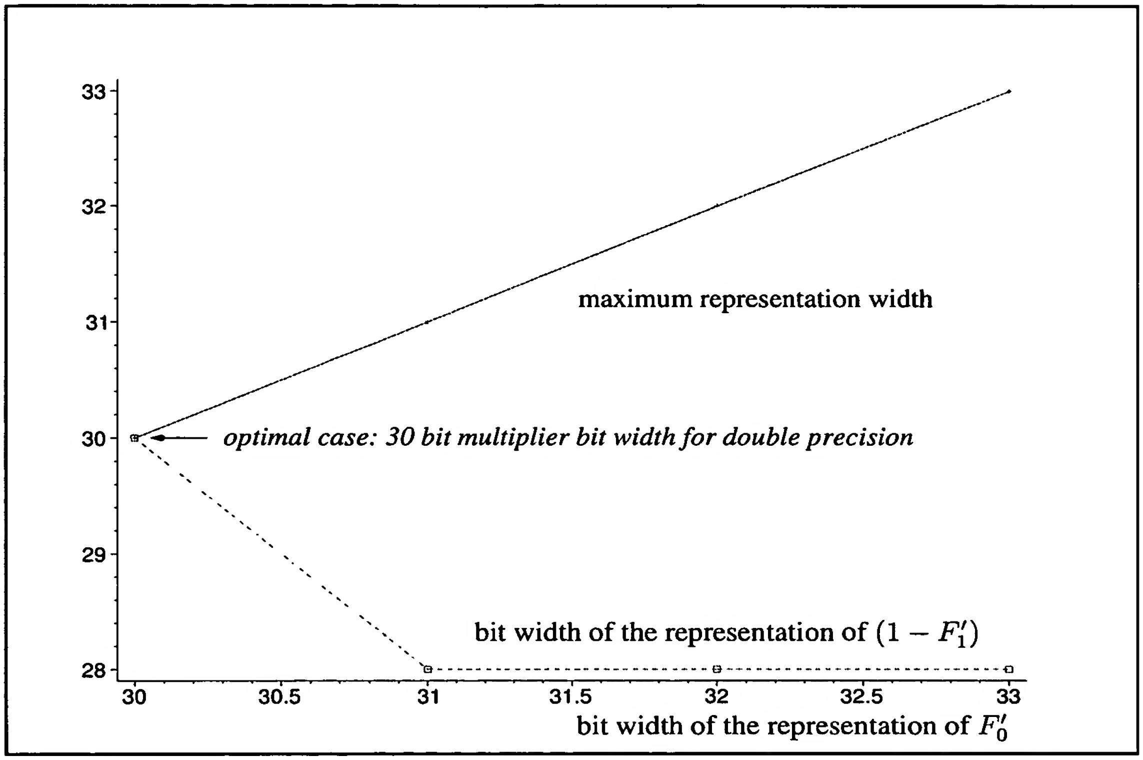

Figure A.3: Feasible and optimized multiplier widths (recoded operand) for double precision implementation.

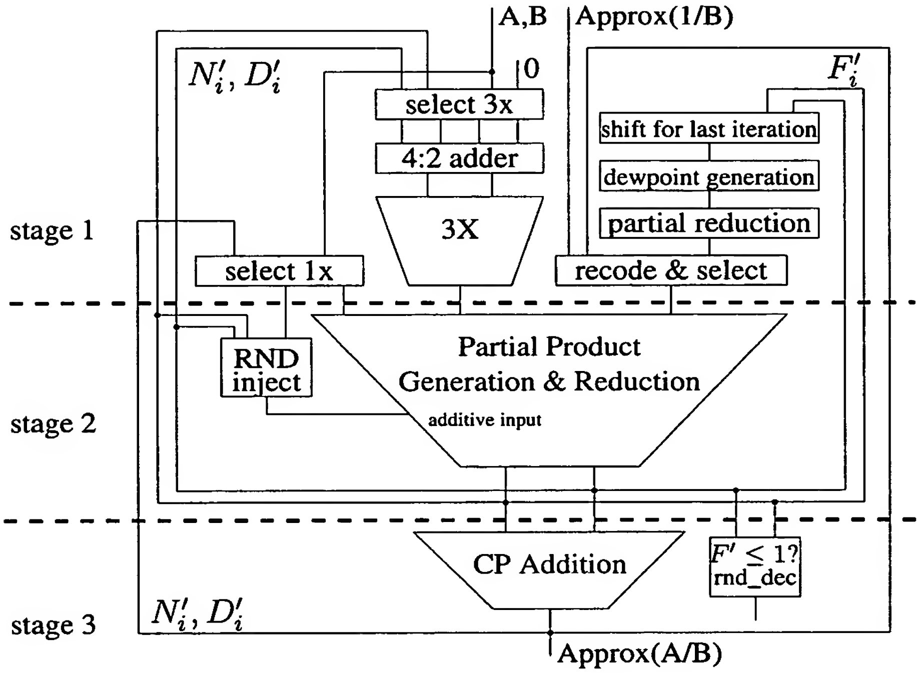

Figure A.4: Microarchitecture for Division Implementation.

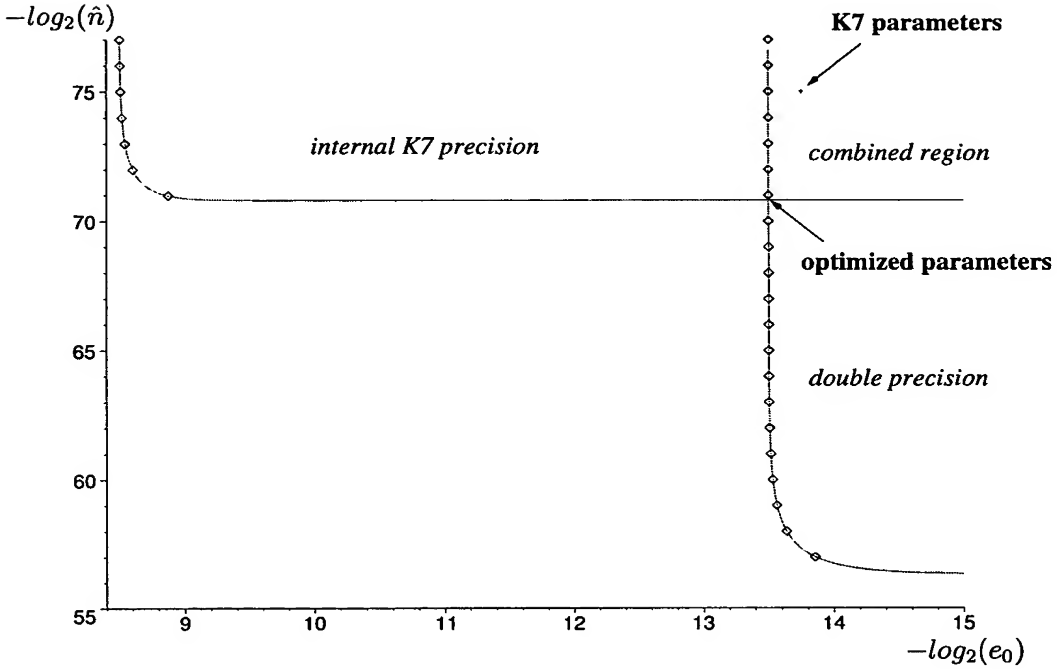

Figure A.5: Feasible parameter combinations of e_0 and $\hat{n}$ for double precision and extended double precision based on Setting I.

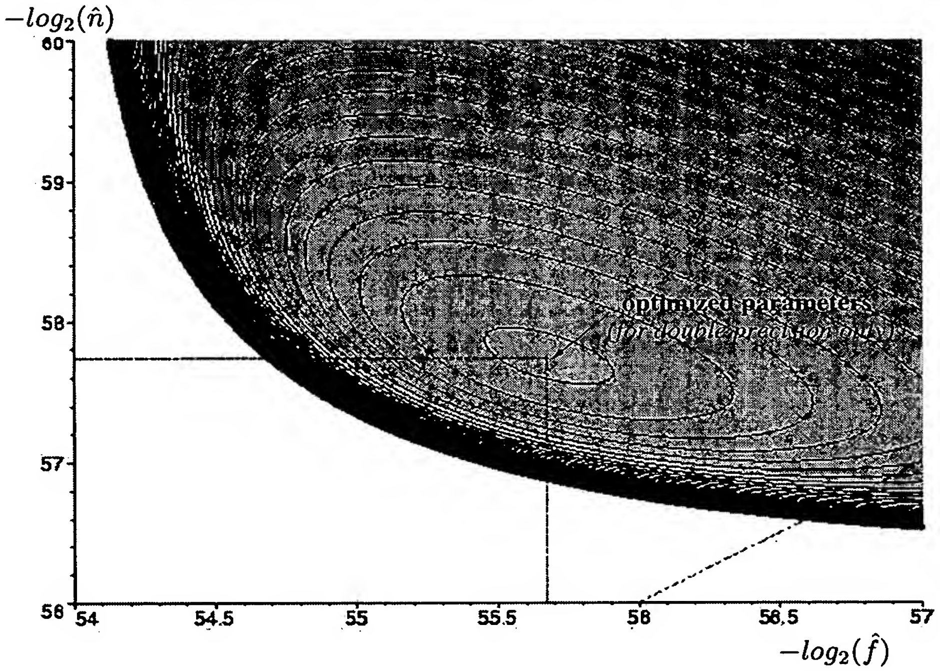

Figure A.6: Feasible $(\hat{n}, \hat{f})$ pairs using Setting IV for double precision are located in the shaded region above the curve. The closed curves depict pairs that lead to designs with equal costs. The central point depicts a feasible pair that leads to the cheapest design.

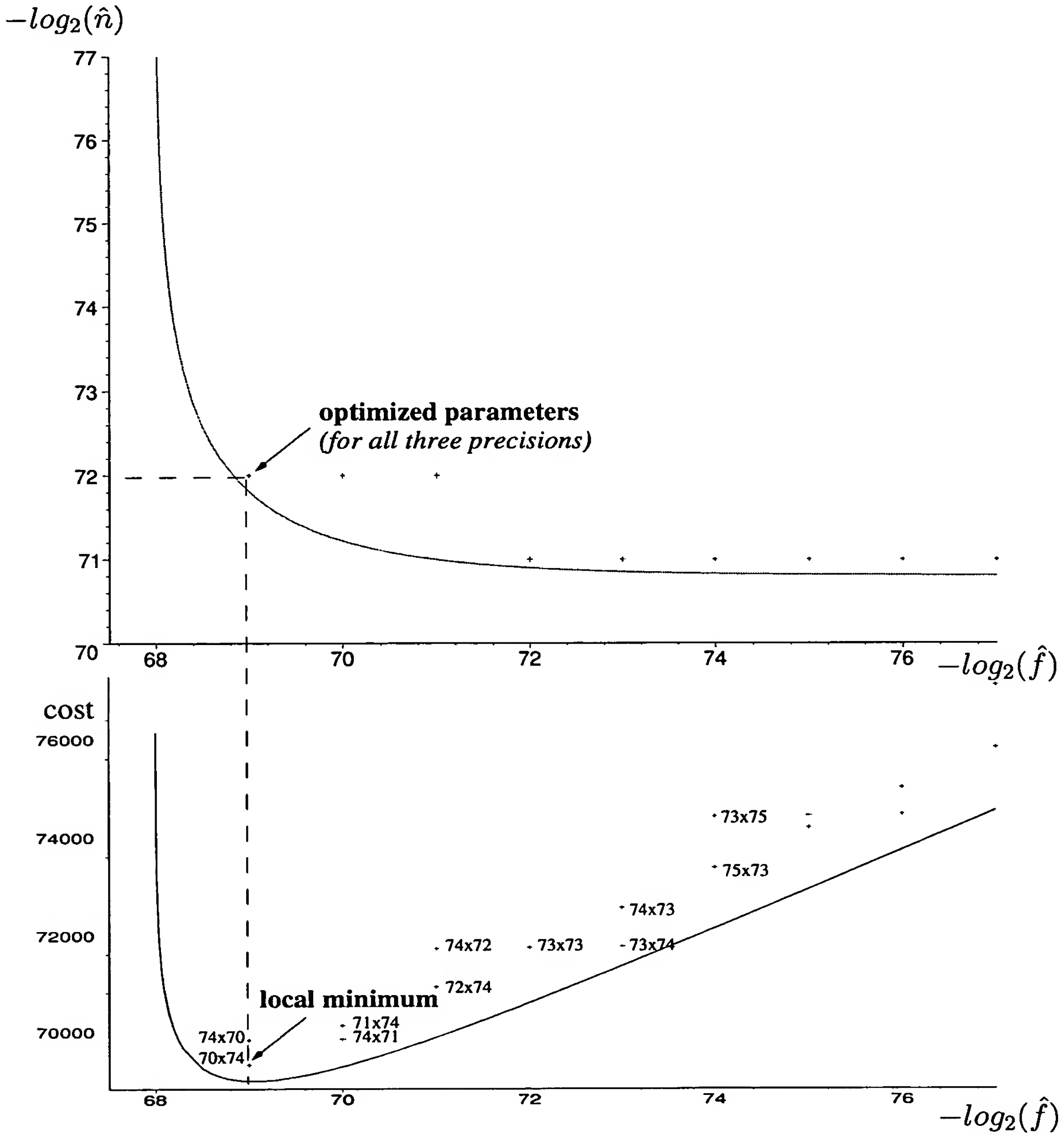

Figure A.7: Top: Feasible $(\hat{n}, \hat{f})$ pairs using setting IV for 68-bit precision (when $e_0 = 2^{-13.51}$). Bottom: Cost of design as a function of $\hat{f}$.

Appendix B

Pipelined Multiplicative Division with IEEE Rounding

abstract

We propose optimized pipelined implementations for Goldschmidt's division algorithm with IEEE rounding based on Booth radix-8 multiplication. The considered optimizations for the quotient approximation are based on a careful general analysis of tight error bounds for the implementation and are accompanied by the utilization of redundant representations, partial compressions, injection-based rounding and rectangular multipliers for the internal computations. For the implementation of IEEE rounding, we introduce the concept of dewpoint rounding, that allows efficient implementation and reduced requirements for the quotient approximation.

On this basis we propose the implementation of different versions of Goldschmidt's division algorithm with different pipeline depths. None of these implementations requires a full-sized multiplier at any stage of the computations. In this way we reduce latency, cost, and enable increased throughput at a reasonable cost. We suggest a full range of pipelining depths: On one extreme is a 3-stage pipeline with a restart time that simply equals the latency minus the number of pipeline stages. On the other extreme is a fully pipelined design.

B.1 Introduction and Summary

Floating point multiplication and division in contemporary microprocessors is implemented by dedicated hardware. Moreover, designs based on multiplicative division contain a full precision multiplier, a rather costly circuit even in the case that it is shared for FP multiplication. In the top part of Table B.1, we list the latencies (i.e., number of cycles required to complete an instruction) and restart times (i.e., number of cycles that elapse till a functional module can engage in a new independent computation) of floating point division modules in several microprocessors. One can readily see that floating point division is rather slow compared to addition and multiplication.

In this paper, we present implementations of Goldschmidt's floating point division algorithm [16] with significantly reduced latency and cost. We consider two settings for floating point division with a precision of p bits. These settings are characterized by the precision of the initial approximation (e.g., lookup table) and the size of the multiplier.

- In the first setting, the precision of the initial approximation is roughly $p/2$ bits, and the multiplier is "full-sized" (i.e. the dimensions of the multiplier are roughly $p \times p$). In the first setting, the latency is 9 clock cycles.
- In the second setting, the precision of the initial approximation is roughly $p/4$ bits, and the multiplier is "half-sized" (i.e. the dimensions of the multiplier are roughly $p \times p/2$). In the second setting, the latency is 11 clock cycles.

We propose implementations for both single precision (i.e., $p_s = 24$) and double precision (i.e., $p_d = 53$). The precision of the initial approximation is roughly $p_d/4 \approx p_s/2$. The multiplier dimensions are roughly $p_d \times p_d/2 \approx 2p_s \times p_s$. This implementation corresponds to the first setting with respect to single precision, and to the second setting with respect to double precision.

The critical path in the designs we propose is a 62 × 30-bit rectangular multiplier that resides in the second pipeline stage. In fact, the multiplier is a radix-8 Booth multiplier, and the Booth digits of the multiplier are precomputed in the first pipeline stage. Hence, the reduction in latency is twofold: (i) fewer clock cycles and (ii) shorter clock periods due to a shorter critical path.

We propose four designs with different restart-times (but with the same latency and critical path). On one extreme, we suggest a three-stage pipeline with a restart-time that simply equals the latency minus the number of pipeline stages. On the other extreme, we suggest a fully pipelined design (i.e., restart-time of one cycle). These designs suggest a reasonable tradeoff between cost and throughput that allows for a full range of restart-times. In the lower part of Table B.1, we list the latencies and restart-times of the four designs we propose.

Both extremes are quite practical. For example, in IEEE double precision, the fully pipelined design contains five rectangular half-sized multipliers (i.e., size 62 × 30) and two quarter-sized multipliers (i.e., size 62 × 15). The number of partial products associated with these multipliers is roughly 3.6 times the number of partial products associated with a full-sized multiplier (i.e. 56 × 56). On the other hand, the three-stage pipeline design contains only one half-sized multiplier.

Related work. Goldschmidt [16] developed a floating point division algorithm in which each iteration requires two independent multiplications. The algorithm maintains three numbers N_i (which converges to the quotient), D_i (which converges two 1), and $F_i = 2 - D_i$ (called the *scaling factor*). The algorithm converges quadratically. Loosely speaking, this means that the number of bits needed to represent F_i doubles in each iteration. This leads to the motivation of a fully pipelined design using a sequence of multipliers (one or two per iteration), where the second operand's length doubles in this sequence.

Probably the main hurdle in implementing Goldschmidt's algorithm is analyzing the errors due to imprecise intermediate computations. InAppendix C a general error analysis of a version of Goldschmidt's algorithm is presented. The advantage of the error analysis is demonstrated by analyzing a micro-architecture similar to that in [28] and showing that a smaller multiplier could be used. An error analysis similar to that in Appendix C enables one to realize the idea of a sequence of multipliers (one per iteration), where only the last multiplier is "full-sized". We further carry an idea of Goldschmidt utilizing the fact that the scaling factor tends to 1 so that we can even make the last multiplier "half-sized".

Techniques. The first step in this paper is an extension of the error analysis in Appendix C (see Sec. B.3). The main reason for extending the analysis is the usage of redundant representations (e.g. carry-save) for intermediate results. Redundant representation incurs two problems: (i) an increase of relative errors due to truncation (whichAppendix C easily handles), and (ii) inability to determine prior to compression if an intermediate result is less than (or greater than) 1.

To reduce the drifting of intermediate results, we suggest to guarantee that the scaling factor F_i (which tends to 1) never falls below 1. For this purpose we introduce an assumption which we call *saturated rounding* of F_i. Saturated rounding (and the fact that the scaling factor tends to 1) enables us to use half the precision in the last iteration. This means that in the last iteration, instead of using a full-sized multiplier (or two passes through a half-sized multiplier), we only require a single pass through a half-sized multiplier. The reason that this methods works is that, as the scaling factor tends to one from above, there will be a run of zeros to the right of the binary point in the binary representation of the scaling factor. We employ our error analysis to prove an upper bound on the scaling factor in the last iteration. This enables us to prove a lower bound on the length of the run of zeros in the binary representation of the scaling factor F_i. We note that a similar observation (but without the one-sided convergence) was made by Goldschmidt [16].

Another contribution of this paper is a new rounding procedure for IEEE floating point division (see Sec. B.4). We refer to this rounding procedure as *dewpoint rounding*. The procedure relies on an error range of the quotient that allows for only two candidates for the final IEEE rounded result. We associate with each candidate r a rounding interval I_r. The rounding interval I_r is simply the set of numbers that are rounded to r. The number that separates the two rounding intervals is called the *dewpoint*. The rounding decision is obtained by comparing the dewpoint and the exact quotient by applying back-multiplication. We present a unified dewpoint rounding procedure for all rounding modes and avoid rounding tables. Previous approaches for IEEE rounding work as follows: (i) Compute a rounding representative of the exact quotient (i.e. a number that belongs to the same rounding interval that the exact quotient belongs to) (ii) Round the representative.

We employ additional techniques, among them: (i) A Booth recoded multiplier that can be fed by either non-redundant binary operands or by redundant carry-save operands [7]. This technique when applied to a Booth radix-8 multiplier enables reducing the feedback latency to two cycles. (ii) Injection based rounding is used to implement directed rounding of intermediate results [9]. (iii) Back-multiplication (i.e comparison between the dewpoint and the exact quotient) requires a full precision multiplication. In the design that uses a half-sized multiplier, two passes through the multiplier are required. To avoid wasting a cycle in preparing the exact dewpoint, we use an approximation of the dewpoint in the first pass (this reduces the latency because, otherwise, we would insert a bubble during this cycle into the pipeline). In the second pass, a correction is used, so that the error caused by the first pass is fully corrected. This method of using an approximate dewpoint heavily relies on the error analysis.

Organization. In Section B.2, notation is introduced, division is defined, and the version of Goldschmidt's algorithm from Appendix C is reviewed. An error analysis of this division algorithm appears in Section B.3. This analysis builds on the analysis that appears in Appendix C. In Section B.4, dewpoint rounding is presented. In Section D.6, we present an implementation of the division algorithm that is based on a full-sized multiplier. For the sake of concreteness, the presentation is for single precision and the implementation uses a double-precision micro-architecture. This means that the half-sized multiplier for double precision is a full-sized multiplier for single precision. In Section B.6, we present an implementation of the devision algorithm that is based on a half-sized multiplier. For the sake of concreteness, an implementation for double precision is presented. In Section B.7, we list various techniques employed to save cost and delay in a hardware implementation. In Section B.8, we consider three pipelining options ranging from a fully pipelined design to a design that reduces the restart-time by half.

	latency				
processor	ALU	FP add	FP mult	FP div single	FP div double
ULTRA-Sparc 3	1	4(1)	4(1)	17(15)	20(18)
Pentium 3	1	3(1)	5(2)	17(17)	32(32)
Pentium 4	1	5(1)	7(2)	23(23)	38(38)
Itanium	1	5(1)	5(1)	30+(11)*	40+(13)*
AMD Athlon	1	4(1)	4(1)	16(13)	20(17)
Power3	1	4(1)	4(1)	17(13)	21(17)
Motorola G4	1	5(1)	5(1)	21(21)	35(35)
Alpha 21064	1	4(1)	4(1)	34(34)	63(63)
Alpha 21164	1	4(1)	4(1)	19(19)	31(31)
Alpha 21264/21364	1	4(1)	4(1)	12(9)	15(12)
R8000	1	4(1)	4(1)	14(11)	20(17)
R12000	1	2(1)	2(1)	14(12)	21(19)
Proposed Divider (1/2)	-	-	-	9(6)	11(8)
Proposed Divider (1)	-	-	-	9(3)	11(4)
Proposed Divider (2)	-	-	-	9(1)	11(2)
Proposed Divider (3)	-	-	-	9(1)	11(1)

Table B.1: Latencies(restart-times) of floating-point operations in current commercial microprocessors compared to the proposed division implementations.

B.2 Preliminaries

Notation Let $x_i x_{i+1} \cdots x_j \in \{0,1\}^*$ denote a binary string. We often denote this string also by $x[i : j]$. We also sometimes refer to x_i as $x[i]$. Since we deal with fractions (mostly in the binade $[1,2)$), the weight associated with the bit x_i is 2^{-i}. Namely, a fraction is represented by the binary digit string $x_0.x_1x_2 \cdots x_{p-1}$.

Let $\sigma = \sigma_i \sigma_{i+1} \cdots \sigma_j \in \{0,1,2\}^*$ denote a carry-save encoded digit string (i.e. $\sigma_i \in \{0,1,2\}$). A carry-save encoded digit string $x[i : j]$ is represented by two binary vectors $xc[i : j]$ and $xs[i : j]$. Each carry-save digit $x[\ell]$ satisfies $x[\ell] = xc[\ell] + xs[\ell]$. We refer to xc (resp. xs) as the *carry-string* (resp. *sum-string*) of x.

Given a binary string $x[i,j]$ and a carry-save encoded digit string $\sigma[i : j]$, we denote the values represented by these string by $|x[i : j]|$ and $|\sigma[i : j]|$, respectively.

Division Operation We consider the following task which captures the main difficulty in IEEE floating-point division. The dividend and the divisor are represented by p-bit binary strings $A[0 : p-1]$ and $B[0 : p-1]$, where $|A|, |B| \in [1, 2)$. Our goal is to compute $Q[0 : p-1]$ that is the binary encoding of the rounded value of the normalized quotient. More formally, (i) Let $A' = A$ if $|A| \geq |B|$, and $A' =$ leftshift(A) if $|A| < |B|$. This way, the quotient $|A'|/|B| \in [1, 2)$. (ii) let $q = |A'|/|B|$, (iii) round q (according to the IEEE standard). The binary string $Q[0 : p-1]$ should satisfy $|Q| = q$.

We consider two precisions: double precision in which $p_d = 53$ and single precision in which $p_s = 24$. Four rounding modes are defined in the IEEE standard: round towards zero round towards zero (RZ), round to nearest even (RNE), round towards infinity (RI) and round towards minus infinity(RMI).

Goldschmidt's division algorithm InAppendix C a variation of Goldschmidt's division algorithm based on directed rounding is presented and analyzed. The algorithm is listed below as Algorithm 2. The multiplications involved and their dependencies are depicted in figure B.1.

Directed roundings are used for all intermediate calculations. For example, N'_i is obtained by rounding down the product $N'_{i-1} \cdot F'_{i-1}$. We denote by n_i the relative error incurred when $N'_{i-1} \cdot F'_{i-1}$ is rounded down. Rounding down translates to the assumption that $n_i \geq 0$. Similarly, rounding down is used for computing F'_i and rounding up is used for computing D'_i. Our error analysis requires that: (a) the operands are in the range A', B satisfy $A' \in [B, 2B)$ and $B \in [1, 2)$, so that $A'/B \in [1, 2)$ (b) all the relative errors incurred by directed rounding are at most $1/4$, and (c) we require that $|e_0| + 3d_0/2 + f_0 < 1/2$. (Note that neither of these requirement imposes any restriction and is very easily met.)

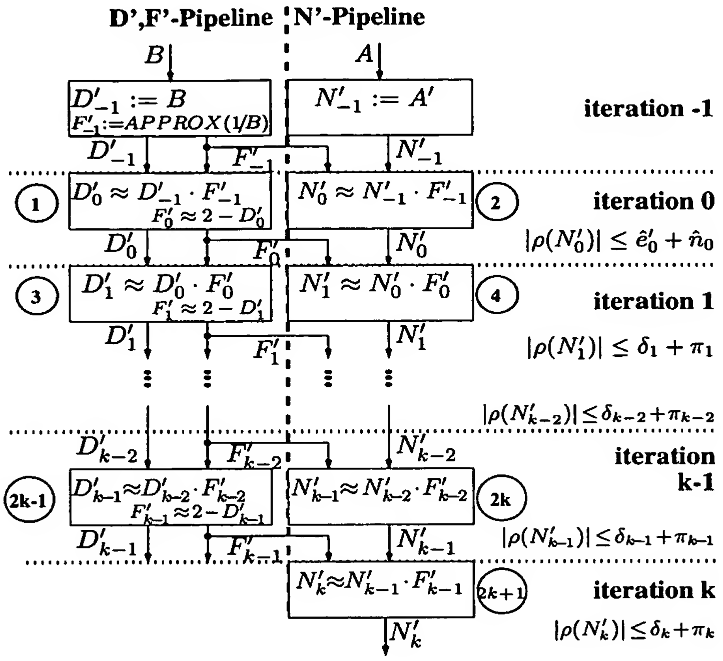
D',F'-Pipeline
N'-Pipeline
B
A
$D'_{-1} := B$
$F'_{-1} := APPROX(1/B)$
$N'_{-1} := A'$
iteration -1
D'_{-1}
F'_{-1}
N'_{-1}
1
$D'_0 \approx D'_{-1} \cdot F'_{-1}$
$F'_0 \approx 2 - D'_0$
$N'_0 \approx N'_{-1} \cdot F'_{-1}$
2
iteration 0
$|\rho(N'_0)| \leq \hat{e}'_0 + \hat{n}_0$
D'_0
F'_0
N'_0
3
$D'_1 \approx D'_0 \cdot F'_0$
$F'_1 \approx 2 - D'_1$
$N'_1 \approx N'_0 \cdot F'_0$
4
iteration 1
D'_1
F'_1
N'_1
$|\rho(N'_1)| \leq \delta_1 + \pi_1$
$|\rho(N'_{k-2})| \leq \delta_{k-2} + \pi_{k-2}$
D'_{k-2}
F'_{k-2}
N'_{k-2}
iteration k-1
2k-1
$D'_{k-1} \approx D'_{k-2} \cdot F'_{k-2}$
$F'_{k-1} \approx 2 - D'_{k-1}$
$N'_{k-1} \approx N'_{k-2} \cdot F'_{k-2}$
2k
D'_{k-1}
F'_{k-1}
N'_{k-1}
$|\rho(N'_{k-1})| \leq \delta_{k-1} + \pi_{k-1}$
$N'_k \approx N'_{k-1} \cdot F'_{k-1}$
2k+1
iteration k
N'_k
$|\rho(N'_k)| \leq \delta_k + \pi_k$

Figure B.1: Schedule of the Iterations of Goldschmidt's division algorithm using approximate arithmetic and an initial approximation for $1/B$. The numbers in circles indicate the sequence of the multiplications involved. For our implementation a bound on the relative error $\rho(N'_i)$ of iteration i appears in each iteration.

B.3 Extension of error analysis

In this section we present a variation of the error analysis presented in Appendix C. The error analysis presented in Appendix C used a simplifying assumption called *strict directed rounding* (SD-rounding). Satisfying SD-rounding is easy if intermediate results are represented in non-redundant binary representation. In our division algorithm redundant representation is used for intermediate results (i.e. carry-save and borrow-save encoded digit strings). Hence we cannot rely on the SD-rounding assumption.

The analysis uses the same requirements used before, namely: (a) the operands A' and B satisfy $A' \in [B, 2B)$ and $B \in [1, 2)$, (b) all the relative errors incurred by directed rounding are at most $1/4$, and (c) $|e_0| + 3d_0/2 + f_0 < 1/2$.

Algorithm 2 Goldschmidt-Approx-Divide(A', B) - Goldschmidt's division algorithm using approximate arithmetic

```
Initialize: $N'_{-1} \leftarrow A'$, $D'_{-1} \leftarrow B$, $F'_{-1} \leftarrow \frac{1-e_0}{B}$.
for $i = 0$ to $k$ do
    $N'_i \leftarrow (1 - n_i) \cdot N'_{i-1} \cdot F'_{i-1}$.
    $D'_i \leftarrow (1 + d_i) \cdot D'_{i-1} \cdot F'_{i-1}$.
    $F'_i \leftarrow (1 - f_i) \cdot (2 - D'_i)$.
end for
Return($N'_i$)
```

B.3.1 Rounding assumptions

The strict-directed rounding assumption is defined as follows.

Assumption 12 **(SD-rounding)** *Directed rounding up (resp. down) r is* strict *if $x < 1$ implies $r(x) \leq 1$ (resp. $x > 1$ implies $r(x) \geq 1$).*

Instead of assuming that SD-rounding holds, we use the following assumptions for iterations $i > 0$:

Assumption 13 **(saturated rounding)** *For $i > 0$,*

$$F'_i = \max\{1, (1 - f_i) \cdot (2 - D'_i)\}.$$

Note that the definition of the relative error f_i in the computation of F'_i is now ambiguous (i.e. suppose $F'_i = 1$ and $(1 - f_i) \cdot (2 - D'_i) < 1$). To avoid ambiguity, define the relative error f'_i as follows:

$$f'_i = \begin{cases} f_i & \text{if } (1 - f_i) \cdot (2 - D'_i) \geq 1 \\ \frac{2 - D'_i - 1}{2 - D'_i} & \text{otherwise.} \end{cases}$$

Note that if $D'_i \leq 1$, then $f_i \geq f'_i \geq 0$. This means that saturated rounding reduces the relative error in the computation of F'_i when $D'_i \leq 1$.

The following assumption requires that multiplication by one is precise.

Assumption 14 **(precise multiplication by** 1**)** *If $F'_i = 1$, then $D'_{i+1} = D'_i$ and $N'_{i+1} = N'_i$.*

We say that the algorithm is *stuck* starting with iteration i if $D'_j = D'_i$ and $N'_j = N'_i$, for every $j \geq i$. Note however, that $F'_i = 1$ does not imply that the algorithm is stuck starting with iteration i. The algorithm may not be stuck if $D'_i < 1$ and eventually the relative error f_j is small enough so that $F'_i > 1$. However, if $D'_i \geq 1$, then the algorithm is stuck starting with iteration i.

We denote the intermediate values obtained with saturated rounding and precise multiplication by 1 also by N'_i, D'_i, and F'_i.

B.3.2 Analysis

Our analysis shows that our weaker assumptions do not affect the bounds on the errors fromAppendix C.

Theorem 15 *For every $i > 0$, the relative error $\rho(N'_i) = \frac{A'/B - N'_i}{A'/B}$ satisfies*

$$0 \leq \rho(N'_i) \leq \pi_i + \delta_i, \tag{B.1}$$

where π_i is defined by $\pi_i \triangleq 1 - (1 - n_i) \cdot \prod_{j=0}^{i-1} \frac{1-n_j}{1+d_j} \geq 0$ and where δ_i is defined by

$$\delta_i := \begin{cases} |e_0| + 3d_0/2 & \textit{for } i = 0 \\ \delta_{i-1}^2 + f_{i-1} & \textit{otherwise} \end{cases}$$

for $i \geq 0$.

Lemma 16 *The following three statements hold:*

a) If $F'_{i-1} = (1 - f_{i-1}) \cdot (2 - D'_{i-1})$ and $D'_{i-1} \in [1 - \delta_{i-1}, 1 + \delta_{i-1}]$, then $D'_i \geq 1 - \delta_i$.

b) If $F'_{i-1} = (1 - f_{i-1}) \cdot (2 - D'_{i-1})$ then $D'_i \leq 1 + d_i$.

c) If $F'_{i-1} = 1 > (1 - f_{i-1}) \cdot (2 - D'_{i-1})$ and $D'_{i-1} \in [1 - \delta_{i-1}, 1]$, then $D'_i \in [1 - \delta_i, 1]$.

Proof: Let $t_j = D'_j - 1$. We first prove Parts a) and b):

$$\begin{aligned} D'_i &= (1+d_i) \cdot \underbrace{(1+t_{i-1})}_{D'_{i-1}} \cdot \underbrace{(1-t_{i-1}) \cdot (1-f_{i-1})}_{F'_{i-1}} \\ &= \underbrace{(1+d_i)}_{\geq 1} \cdot \underbrace{(1-t^2_{i-1})}_{\leq 1} \cdot \underbrace{(1-f_{i-1})}_{\leq 1} \end{aligned}$$

It follows that $D'_i \geq (1 - \delta^2_{i-1} - f_{i-1}) = 1 - \delta_i$, which implies Part a). It also follows that $D'_i \leq 1 + d_i$, which implies Part b).

c) Since $F'_{i-1} = 1$, precise multiplication by 1 implies that $D'_i = D'_{i-1}$, so all we need to show is that $D'_i > 1 - \delta_i$. Since $1 > (1 - f_{i-1}) \cdot (2 - D'_{i-1})$, we can write:

$$\begin{aligned} D'_i &> (1 - f_{i-1}) \cdot (2 - D'_{i-1}) \cdot D'_{i-1} \\ &\geq (1 - f_{i-1}) \cdot (1 - \delta^2_{i-1}) \\ &\geq 1 - \delta_i, \end{aligned}$$

and Part c) follows. □

Proof of Theorem 15: InAppendix C Equation (B.1) is proven for iteration i whenever $D'_{i-1} \in [1 - \delta_{i-1}, 1 + \delta_{i-1}]$ and $F'_{i-1} = (1 - f_{i-1}) \cdot (2 - D'_{i-1})$. In fact, these two conditions are used in Appendix C to prove that $1 - \delta_i \leq D'_{i-1} \cdot F'_{i-1} \leq 1$, which implies Equation (B.1) for iteration i. We rely on the proof presented in Appendix C and deal here with the effect of saturated rounding.

Let i_D and i_F be defined as follows:

$$\begin{aligned} i_D &\triangleq \min\{i > 0 : D'_i > 1\} \\ i_F &\triangleq \min\{i > 0 : (1 - f_i) \cdot (2 - D'_i) < 1\} \end{aligned}$$

Note that if $D'_i > 1$, then $(1 - f_i) \cdot (2 - D'_i) < 1$, and hence, $i_F \leq i_D$.

We need to consider the 3 cases: (i.) $i \leq i_F$; (ii.) $i_F < i \leq i_D$; and (iii.) $i > i_D$. Note that case (ii.) only occurs if $i_F \neq i_D$.

Case (i.): One can directly verify that $1 - \delta_0 \leq D'_0 \leq 1 + \delta_0$. Since saturated rounding does not take place in iteration $i = 0$, it follows that $F'_0 = (1 - f_0) \cdot (2 - D'_0)$. Hence the error analysis in Appendix C implies that Equation B.1 holds for $i = 1$.

For $0 < i - 1 < i_F$ we have $D'_{i-1} \leq 1$ and $F'_{i-1} = (1 - f_{i-1}) \cdot (2 - D'_{i-1}) \geq 1$ since $i - 1 < min(i_D, i_F)$. By applying induction and Part a) of Lemma 16, it follows that $D'_{i-1} \in [1 - \delta_{i-1}, 1]$. Hence the proof in Appendix C applies, and Equation B.1 holds for i. It follows that Equation B.1 holds for every $i \in [1, i_F]$.

Case (ii.): By applying induction and Parts a) and c) of Lemma 16, it follows that $D'_{i-1} \in [1 - \delta_{i-1}, 1]$, for every $i - 1 < i_D$. If $F'_{i-1} = (1 - f_{i-1}) \cdot (2 - D'_{i-1})$, then Equation (B.1) holds for i. If $F'_{i-1} = 1 > (1 - f_{i-1}) \cdot (2 - D'_{i-1})$, then by Part c) of Lemma 16, $D'_i \in [1 - \delta_i, 1]$. By precise multiplication by 1 it follows that $D'_i = D'_{i-1}$, and hence $1 - \delta_i \leq D'_{i-1} \cdot F'_{i-1} \leq 1$. Hence the error analysis in Appendix C is applicable, and Equation (B.1) holds for $i \leq i_D$.

Case (iii.) ($i > i_D$)**:** Note that the algorithm is stuck starting from iteration i_D. Since $\pi_{j+1} \geq \pi_j$, it suffices to prove that $0 \leq \rho(N'_{i_D}) \leq \pi_{(i_D+1)}$.

We now prove that $D_{i_D} \in [1, 1 + d_{i_D}]$. The lower bound follows from the definition of i_D. Part b) of Lemma 16 proves that if $F_{i_D-1} = (1 - f_{I_D-1}) \cdot (2 - D_{I_D-1})$, then $D_{i_D} \leq 1 + d_{i_D}$. So all we need to prove is that $F'_{i_D-1} = (1 - f_{i_D-1}) \cdot (2 - D'_{i_D-1})$. Since $D_{i_D} > 1$ it follows that if $D_{i_D-1} \leq 1$, then $F'_{i_D-1} \neq 1$, hence $F'_{i_D-1} = (1 - f_{i_D-1}) \cdot (2 - D'_{i_D-1})$. The minimality of i_D implies that if $D_{i_D-1} > 1$, then $i_D = 1$. But saturated rounding is not used in iteration $i = 0$, so the upper bound on D'_{i_D} holds.

We now expand $\frac{N'_j}{D'_j}$ as follows:

$$\begin{aligned}\frac{N'_j}{D'_j} &= \frac{N'_{j-1}}{D'_{j-1}} \cdot \frac{1-n_j}{1+d_j} \\ &= \frac{A}{B} \cdot \prod_{\ell=0}^{j} \frac{1-n_\ell}{1+d_\ell}.\end{aligned}$$

Thus,

$$N'_j = \frac{A}{B} \cdot \prod_{\ell=0}^{j} \frac{1-n_\ell}{1+d_\ell} \cdot D'_j$$

and

$$\rho(N'_j) = 1 - \prod_{\ell=0}^{j} \frac{1-n_\ell}{1+d_\ell} \cdot D'_j. \tag{B.2}$$

Since $D'_{i_D} \geq 1$, it follows that

$$\begin{aligned}\rho(N'_{i_D}) &\leq 1 - \prod_{\ell=0}^{i_D} \frac{1-n_\ell}{1+d_\ell} \\ &\leq 1 - (1 - n_{(i_D+1)}) \prod_{\ell=0}^{i_D} \frac{1-n_\ell}{1+d_\ell} \\ &= \pi_{(i_D+1)}\end{aligned} \tag{B.3}$$

which gives the desired upper bound on $\rho(N'_{i_D})$. To prove the lower bound we use $D'_{i_D} \leq 1 + d_{i_D}$ with equation (B.2) as follows:

$$\begin{aligned}\rho(N'_{i_D}) &\geq 1 - (1 + d_{i_D}) \prod_{\ell=0}^{i_D} \frac{1-n_\ell}{1+d_\ell} \\ &= 1 - (1 - n_{i_D}) \prod_{\ell=0}^{(i_D-1)} \frac{1-n_\ell}{1+d_\ell} \\ &= \pi_{i_D} \geq 0.\end{aligned}$$

This completes the proof for the last of the three cases. □

Corollary 17 *For every* $i > 0$, $F_i' \in [1, 1 + \delta_i]$

Proof: It follows from Assumption 13 (saturated rounding) that $F_i' \geq 1$. We only need to deal with the case that $F_i' = (1 - f_i) \cdot (2 - D_i')$. In the proof of Theorem 15 we showed that $D_i' \geq 1 - \delta_i$, for $i > 0$. Therefore, $F_i' = (1 - f_i) \cdot (2 - D_i') \leq 1 + \delta_i$. □

B.4 IEEE rounding of the quotient

In this section we describe how IEEE rounding is done in our algorithm. We present a novel method called *dewpoint rounding*.

B.4.1 Background on IEEE rounding

The IEEE Floating-Point Standard 754 [17] requires the implementation of all four IEEE rounding modes for all basic arithmetic operations and for all precisions supported. IEEE rounding is supposed to be computed on the exact result of the operation (which is A'/B in our case). For most operations the exact result is not available or too expensive to be computed. It is therefore common practice to compute the IEEE rounding result by computing a rounding representative first and compute IEEE rounding on the representative that leads provably to the same result [10].

In the case of multiplicative division it is even difficult to find a representative quotient from an approximation of the quotient. In [18], it is proven that the length of runs of zeros (or ones) in the binary representation of the exact quotient is limited. This means that a rounding representative can be decided from an approximate quotient if the precision of the approximate quotient is roughly twice the target precision. We do not follow this approach since it involves one additional iteration for multiplicative division algorithms, adds significant delay to the implementation, and requires very wide intermediate operand representations for the last iteration.

Back-multiplication is the common method for computing the IEEE rounded quotient. In this method, the approximate quotient is multiplied by the divisor B, and then this product is compared with the dividend A' [19, 34].

In the sign-magnitude representation of floating-point numbers, the four IEEE rounding modes can be reduced to three rounding modes RZ, RI, and RNU (round to nearest up) based on the numbers' sign [32].

Observation 18 *The exact quotient A'/B cannot be a midpoint between two representable numbers. Therefore RNU and RNE are equivalent rounding modes with respect to division.*

Based on the above reductions and Observation 18, we only need to consider the three rounding modes RZ, RI, and RNU.

Injection based rounding [9] was introduced to further simplify rounding. Injection based rounding reduces these three rounding modes to truncation. This reduction is obtained by adding an injection that only depends on the rounding mode.

B.4.2 Dewpoint Rounding

Dewpoints

The concept of dewpoint rounding is inspired by the following concept of a dewpoint in meteorology: the dewpoint is the threshold temperature that separates between the events of rain and no rain.

If the quotient's estimate is close enough to the accurate un-rounded quotient, then rounding only needs to select between two values. We denote by $RC_1 < RC_2$ the two candidate values for the IEEE rounded result. Note that $RC_2 = RC_1 + 2^{-p+1}$. For each rounding candidate RC_j, we denote by I_j the rounding interval that comprises the pre-image of RC_j with respect to rounding. Namely, I_j is the set of numbers that are rounded to RC_j. The definition of RZ, RNU, and RI rounding implies that since RC_1 and RC_2 are successive representable numbers, the rounding intervals I_1 and I_2 share an endpoint. This common endpoint is called the *dewpoint*.

To guarantee only two rounding candidates the following assumption on N'_k must hold.

Assumption 19 *Prior to the computation of the dewpoint, the quotient's estimate N'_k satisfies:* $0 \leq \rho(N'_k) < 2^{-p}$.

Computation of rounding candidates

Given a carry-save encoding $\sigma[0 : t]$ of N'_k, we wish to compute the dewpoint and the rounding candidates RC_1 and RC_2. A unified computation method based on injections is used for all rounding modes. We begin by dealing with RZ rounding and then reduce the two other rounding modes RNU and RI to RZ.

We use some notation related to the target precision p.

Definition 3 *We refer to multiples of* 2^{-p+1} *as* representable significands *and to odd multiples of* 2^{-p} *as* mid-points.

RZ rounding. Since $0 \leq \rho(N'_k) < 2^{-p}$ and $\frac{|A'|}{|B|} < 2$, it follows that $\frac{|A'|}{|B|} \in [N'_k, N'_k + 2^{-p+1})$. A carry-save encoded digit string $\sigma[0 : t]$ is used to represent N'_k. Let $bin(\sigma)[0 : t]$ denote the binary string that satisfies $|bin(\sigma)[0 : t]| = |\sigma[0 : t]|$. (Note that this notation does not imply full compression of σ in the implementation. We use this notation just to simplify the description of the rounding.)

Truncation of $bin(\sigma)[0 : t]$ after position $p - 1$ yields:

$$|bin(\sigma)[0 : p - 1]| \in (N'_k - 2^{-p+1}, N'_k].$$

Let $r \triangleq |bin(\sigma)[0 : p - 1]|$. We conclude that

$$\frac{|A'|}{|B|} \in [r, r + 2 \cdot 2^{-p+1}). \tag{B.4}$$

In the RZ rounding mode the interval $[r, r + 2 \cdot 2^{-p+1})$ is the union of exactly two rounding intervals: $[r, r+2^{-p+1})$ and $[r+2^{-p+1}, r+2\cdot 2^{-p+1})$. Hence, the dewpoint in this case is $r+2^{-p+1}$. The rounding candidates are simply $RC_1 = r$ and $RC_2 = r + 2^{-p+1}$.

This method fails for RI and RNU since in these rounding modes the interval $[r, r + 2 \cdot 2^{-p+1})$ intersects three rounding intervals.

Reduction of RI and RNU to RZ. We reduce the rounding modes RI and RNU to RZ by using injections [11] as follows.

Let $\text{INJ}_{rnd}(mode)$ denote an injection constant that depends only on the rounding mode an on the precision p. We define $\text{INJ}_{rnd}(mode)$ as follows.

$$\text{INJ}_{rnd}(mode) := \begin{cases} 0 & \text{for } mode = RZ \\ 2^{-p} & \text{for } mode = RNU \\ 2^{-p+1} - 2^{-t} & \text{for } mode = RI. \end{cases} \tag{B.5}$$

For $x \in [1, 2)$, the addition of $\text{INJ}_{rnd}(mode)$ reduces RI and RNU to RZ in the following sense (c.f. [11]):

$$\begin{aligned} RNU(x) &= RZ(x + \text{INJ}_{rnd}(RNU)) \\ RI(x) &= RZ(x + \text{INJ}_{rnd}(RI)). \end{aligned} \tag{B.6}$$

Hence it suffices to compute $RZ(\frac{|A'|}{|B|} + \text{INJ}_{rnd}(mode))$.

We now repeat the analysis with $\text{INJ}_{rnd}(mode)$ as follows. Since $0 \leq \rho(N_k') < 2^{-p}$, it follows that $\frac{|A'|}{|B|} + \text{INJ}_{rnd}(mode) \in [N_k' + \text{INJ}_{rnd}(mode), N_k' + \text{INJ}_{rnd}(mode) + 2^{-p+1})$. Let $bin(\sigma')[0 : t]$ denote the binary string that satisfies

$$|bin(\sigma')[0 : t]| \triangleq |\sigma[0 : t]| + \text{INJ}_{rnd}(mode). \tag{B.7}$$

Truncation of $bin(\sigma')[0:t]$ gives the following estimate:

$$|bin(\sigma')[0:p-1]| \in (N'_k + \text{INJ}_{rnd}(mode) - 2^{-p+1}, N'_k + \text{INJ}_{rnd}(mode)].$$

Let $r' \triangleq |bin(\sigma')[0:p-1]|$. We conclude that

$$\frac{|A'|}{|B|} + \text{INJ}_{rnd}(mode) \in [r', r' + 2 \cdot 2^{-p+1}). \tag{B.8}$$

Definition 4 *The rounding candidates are defined as follows:*

$$RC_1 = r' \qquad \textit{and} \qquad RC_2 = r' + 2^{-p+1}.$$

The following claim summarizes the reduction.

Claim 20 *For every rounding mode mode $\in$ {RZ,RNU,RI}, the IEEE rounded result satisfies the following equation:*

$$RZ\left(\frac{|A'|}{|B|} + \text{INJ}_{rnd}(mode)\right) = \begin{cases} RC_1 & \textit{if } \frac{|A'|}{|B|} + \text{INJ}_{rnd}(mode) < RC_2, \\ RC_2 & \textit{otherwise.} \end{cases} \tag{B.9}$$

Note that the decision does not require division since $\frac{|A'|}{|B|} + \text{INJ}_{rnd}(mode) < RC_2$ if and only if $|A'| < (RC_2 - \text{INJ}_{rnd}(mode)) \cdot |B|$.

Dewpoint back-multiplication

The disadvantage in Claim 20 is in the complexity of computing the comparison $|A'| < (RC_2 - \text{INJ}_{rnd}(mode)) \cdot |B|$. Specifically, in RI mode, the representation of $\text{INJ}_{rnd}(mode)$ requires t bits of precision.

In this section we show how back-multiplication can be done without having to deal with positions to the left of position $[p]$.

By Claim 20, the rounded quotient equals RC_1 if and only if

$$\frac{|A'|}{|B|} < r' + 2^{-(p-1)} - \text{INJ}_{rnd}(mode). \tag{B.10}$$

Since we are interested in simplifying the computation of the dewpoint, we define the dewpoint displacement constant $C_{dew}(mode)$ as follows.

$$C_{dew}(mode) \triangleq \begin{cases} 2^{-(p-1)} & \text{if } mode = \text{RZ} \\ 2^{-p} & \text{if } mode = \text{RNU} \\ 0 & \text{if } mode = \text{RI}. \end{cases}$$

Note that $C_{dew}(mode) \leq 2^{-p+1} - \text{INJ}_{rnd}(mode)$ for every rounding mode.

Definition 5 *The* dewpoint *is defined as follows:*

$$dewpoint \triangleq r' + C_{dew}(mode).$$

Note that $dewpoint = r' + 2^{-(p-1)} - \text{INJ}_{rnd}(mode)$ if the rounding mode is RZ or RNU. If the rounding mode is RI, then dewpoint is too small by 2^{-t}. Since 2^{-t} is the our resolution, it follows that $\frac{|A'|}{|B|} < r' + 2^{-(p-1)} - \text{INJ}_{rnd}(mode)$ if and only if $\frac{|A'|}{|B|} \leq r' + 2^{-(p-1)} - \text{INJ}_{rnd}(mode) - 2^{-t}$. This implies in RI, the rounded quotient equals RC_1 if and only if $\frac{|A'|}{|B|} \leq dewpoint$ (as opposed to the strict inequality in Equation B.10).

Back-multiplication is implemented by computing the sign (positive, zero, negative) of the expression

$$|A'| - |B| \cdot dewpoint.$$

The following claim summarizes how back-multiplication is used to determine the rounded quotient.

Claim 21 *The IEEE rounded quotient Q equals RC_1 or RC_2. The decision is determined as* *follows:*

$$Q = \begin{cases} RC_1 & \mathit{if}\,(|A'| - |B| \cdot \mathit{dewpoint}) < 0 \\ RC_2 & \mathit{if}\,(|A'| - |B| \cdot \mathit{dewpoint}) > 0 \\ RC_1 & \mathit{if}\,(|A'| - |B| \cdot \mathit{dewpoint}) = 0 \text{ and } \mathit{mode} = \mathit{RI} \\ RC_2 & \mathit{if}\,(|A'| - |B| \cdot \mathit{dewpoint}) = 0 \text{ and } \mathit{mode} = \mathit{RZ} \end{cases}$$

The case $\frac{|A'|}{|B|} = \mathit{dewpoint}$ and $\mathit{mode} =$ RNU is omitted since by Observation 18 it cannot occur.

B.5 An implementation with a full sized multiplier

In this section we present an implementation of our algorithm that uses a full sized multiplier (i.e., for a target precision p, the multiplier dimensions are roughly $p \times p$) and an initial approximation that whose precision is roughly $p/2$. To be concrete, we present detailed design for single precision (i.e. $p_s = 24$) that has a latency of 9 cycles.

The main motivation for this setting is for performing single precision divisions with a design built for double precision. We assume that core of the design for double precision division is a half-sized multiplier (i.e., somewhat larger than a $p_d \times (p_d/2)$ multiplier, where $p_d = 53$). Such a multiplier is larger than a $2p_s \times p_s$ multiplier. This implies that, when performing single precision division, the multiplier is not only a full-sized multiplier, but a double-width multiplier. We can employ a double-width multiplier to reduce to latency from 9 cycles to 8 cycles (see item 1 in Section D.6.4).

B.5.1 Basic microarchitecture

A block diagram with the three stages of our basic microarchitecture is depicted in Fig. B.2. The core of this microarchitecture is a radix-8 Booth recoded Multiplier. The microarchitecture allows for feeding of previously computed products back to the multiplier as either operands. Latency is reduced by allowing the feedback to be in redundant representation. The multiplier also supports a multiply-add operation (i.e. $A' \times B + C$). In this section the addend (i.e. the number to be added to the product) is a binary number, so only one row in the adder tree is allocated for the addend.

The multiplication circuitry is divided into 4 pipeline stages. Below we describe some details of the stages:

0. In pipeline stage 0, an estimate for the reciprocal $1/B$ is computed and the values of A and B are compared to determine the pre-normalized value A'. Additional hardware is included for scheduling and making available the operand values A', B and approx$(1/B)$ to the inputs of the first and the second stage.

1. In the first stage, the two operands of the multiplier (i.e., the multiplicand and multiplier) are prepared for the addition of the partial products in the second stage. The second operand (i.e., the multiplier) is recoded in Booth radix-8 digits and the partial products are generated. Following [35], the recoding can accept either a binary string or a carry-save encoded digit string.

   The first operand (i.e., the multiplicand) is processed as follows. The $3\times$ multiple of the multiplicand is computed using an adder. A feedback product, encoded as a carry-save digit string, can be used as a multiplicand as follows. The computation of the $3\times$ multiple is preceded by a 4:2-adder that computes a carry-save encoding of the $3\times$ multiple. This carry-save encoded digit string is compressed to a binary number by the adder. Meanwhile, the binary representation of the multiplicand is computed by the adder in the third pipeline stage. This saves an adder in the first pipeline stage (we need to insure that the adder in the third pipeline stage is indeed available).

2. In the second stage, the partial products are compressed by an adder tree. In addition to the partial products, there is a row that is dedicated for (i) a carry-save round-up rounding injection (see Section D.5.2), (ii) an IEEE rounding injection (see Equation D.6), or (iii) a two's complement number (see Section D.5.3).

3. The third stage contains an adder and a comparator. The adder has the following tasks: (i) compressing the multiplicands from carry-save representation to binary representation (ii) computing RC_1 and RC_2.

   The comparator is used for computing the sign of the back-multiplication and checking for $F_1' < 1$ from the carry-save representation of D_1'. We cannot use the adder as a comparator due to conflicts between two pipelined divisions.

   Additional circuitry is installed in the third stage, including: (i) circuitry for feeding the adder with 2^{-23}, (ii) circuitry for computing RC_1, RC_2, and (iii) circuitry for selecting between RC_1 and RC_2.

B.5.2 Pipeline schedule

Table B.2 lists the schedule of operations that implements our algorithm for single precision based on a "full-sized" multiplier and a "half-sized" initial approximation. In the error analysis presented in Section D.6.3 we show that a 30 × 27 rectangular multiplier together with an error bound of $|e_0| \leq 2^{13.74}$ guarantees correctness. We use these parameters in the detailed description below. We now describe the cycles one by one.

cycle 0: In cycle 0 an initial approximation of $1/B$ is computed. We denote this approximate reciprocal by $(1 - e_0)/B$. Since this reciprocal is recoded in the next cycle, it is possible to have the reciprocal represented in binary or redundant format. Moreover, in cycle 1, very little processing of this reciprocal is required. This means that one could actually allow the reciprocal approximation to take place also during part of cycle 1.

cycle 1: Iteration 0 of the algorithm begins in this cycle. The approximate reciprocal $(1 - e_0)/B$ is assigned to F'_{-1} and is fed as the second operand of the multiplier. The approximate reciprocal is recoded as Booth radix-8 digit string. The first operand of the multiplier D'_{-1} is simply assigned the value B. The $3\times$ multiple of D'_{-1} is computed in this cycle.

In the third pipeline stage, A and B are compared. If $A < B$, then we set $A' \leftarrow 2 \cdot A$, otherwise $A' \leftarrow A$. This is how we guarantee that $A' \in [B, 2B)$. Note that since A' can be in the binade $[2, 4)$, A' has a bit in position $[-1]$.

cycle 2: Two activities take place in this cycle. In the first pipeline stage, $A'[-1 : 28]$ is assigned to N'_{-1} and the $3\times$ multiple of N'_{-1} is computed. In the second pipeline stage the rounded up product $D'_{-1} \cdot F'_{-1}$ is stored in $D'_0[0 : 28]$ as a carry-save number. The rounding up of a result represented in carry-save representation is done by using a rounding up injection particular to carry-save representations $\text{INJCS}_{RU}(28, 28 + 14)$. The details of this method are explained in Section D.5.2.

cycle 3: The carry-save representation of D'_0 is compressed to binary in the third pipeline stage. In the second pipeline stage, the rounded down product $N'_{-1} \cdot F'_{-1}$ is stored in $N'_0[-1 : 28]$. Rounding down is simply achieved by truncation of the carry-save representation. Note that our error analysis (Theorem 15) only implies that $N'_i < 2$ for $i > 0$. Hence we need position $[-1]$ in N'_0.

cycle 4: In the first pipeline stage, a carry-save representation of $F'_0 = 2 - D'_0$ is computed and then Booth recoded. (Recall that D'_0 is compressed to binary in the previous cycle.) In addition, the $3\times$ multiple of N'_0 is computed.

In the third pipeline stage, the carry-save representation of N'_0 is compressed to binary.

cycle 5: In this cycle we compute in the second stage N''_1 in carry-save representation with the value:

$$N''_1 \leftarrow N'_0 \cdot F'_0 + \text{INJ}_{rnd}(mode).$$

The carry-save representation of N''_1 equals the value of $|bin(\sigma')|$ defined in Equation B.7.

cycle 6: In the third pipeline stage, RC_1 is computed by compressing N''_1 and truncating at position 23. The dewpoint equals $RC_1 + c_{dew}(mode)$. However, to reduce delay, we compute the dewpoint as follows. A carry-save representation of the dewpoint is obtained by adding $N_1''[0:23] + c_{dew}(mode) + carry_{23}(N_1''[24:28])$, where $carry_{23}(N_1''[24:28])$ is a bit that denotes if $N_1''[24:28] \geq 2^{-23}$. The dewpoint is Booth recoded and the 3× multiple of B is computed.

cycle 7**:** Back-multiplication takes place in the second pipeline stage. We use a representation called two's complement carry-save representation [7]. In this representation the most significant position has a negative weight. Our goal in the back multiplication is to compute the z-sign of $\alpha = B \cdot dewpoint - A'$. (The z-sign indicates equality, greater than, or less than.) In Section D.5.3 we show that it suffices to examine digits in positions [22 : 47] to determine the z-sign of α.

cycle 8**:** In the third pipeline stage, (i) RC_2 is computed, (ii) the z-sign of the back-multiplication is determined from $\alpha[22:47]$, and (iii) the IEEE rounded quotient is selected from RC_1 and RC_2 according to the rounding mode and the z-sign of α.

B.5.3 Correctness

In this section we prove the correctness of a design based on a rectangular 30 × 27-multiplier and an initial approximation with a relative error $e_0 \leq 2^{-13.74}$. To support single precision ($p_s = 24$), dewpoint rounding requires that Assumption 19 holds. Namely, $0 \leq \rho(N_1') < 2^{-24}$.

The analysis presented in this section is a concrete analysis for single precision. More generally, this analysis applies to the following situation: (i) the accuracy of the initial approximation is roughly $2^{-p/2}$; (ii) the quotient is approximated by N_1' (namely, the algorithm has 1 iteration of quadratical convergence); and (iii) the multiplier is a rectangular multiplier whose dimensions are not smaller than roughly $p_s \times p_s$.

In Table B.2 we assumed that initial approximation $F'_{-1} = \frac{1-e_0}{B}$ is represented by 15 bits. This implies that the multiplier can easily accept F'_{-1} as a short operand.

Claim 22 *If the relative error in initial approximation satisfies* $|e_0| \leq 2^{-13.74}$*, then correct double precision IEEE rounding is obtained with a rectangular* 30×27*-bit multiplier.*

Proof: All we need to show is that $0 \leq \rho(N'_1) < 2^{-24}$. By Theorem 15, $0 \leq \rho(N'_1) \leq \pi_1 + \delta_1$. We expand π_1 and δ_1 to obtain:

$$\begin{aligned} \pi_1 &= 1 - (1 - n_1) \cdot \frac{1 - n_0}{1 + d_0} \\ \delta_1 &= \left(|e_0| + \frac{3}{2} \cdot d_0\right)^2 + f_0. \end{aligned} \tag{B.11}$$

To prove $\pi_1 + \delta_1 < 2^{-24}$, we prove upper bounds on the relative error terms (i.e., n_i, f_i, d_i) and substitute these upper bounds in Equation B.11. We bound these relative error terms using absolute error terms defined as follows:

Definition 6 *The absolute errors of intermediate computations are defined as follows:*

$$\begin{aligned} neps_i &\triangleq n_i \cdot N'_{i-1} \cdot F'_{i-1} \\ deps_i &\triangleq d_i \cdot D'_{i-1} \cdot F'_{i-1} \\ feps_i &\triangleq f_i \cdot (2 - D'_i). \end{aligned}$$

We now bound each of the relative error terms:

Bound on d_0: According to Table B.2, D'_0 is represented by a carry-save encoded digit string with digits in positions $[0 : 28]$. This digit string holds the *rounded up* value of $D'_{-1} \cdot F'_{-1}$. This implies that the absolute error $deps_0$ associated with the computation of D'_0 is in the range $[0, 2 \cdot 2^{-28})$. Since the precise product $D'_{-1} \cdot F'_{-1}$ equals $1 - e_0$, this product is in the range $1 \pm 2^{-13.74}$. It follows that $d_0 < 2^{-27}/(1 - 2^{-13.74}) < 2^{-26.99989}$.

Bound on n_0: As in the case of D'_0, N'_0 is also represented by a carry-save encoded digit string with digits in positions $[0 : 28]$. This digit string holds the *rounded down* value of $N'_{-1} \cdot F'_{-1}$. This implies that the absolute error $neps_0$ associated with the computation of D'_0 is in the range $[0, 2 \cdot 2^{-28})$. The product $A' \cdot F'_{-1}$ equals $\frac{A'}{B} \cdot (1 - |e_0|)$. Since $A' \geq B$, it follows that $A' \cdot F'_{-1} \geq 1 - |e_0|$, and therefore, $n_0 < 2^{-27}/(1 - |e_0|)$.

Bound on f_0: Note that D'_0 is compressed to binary non-redundant representation in the third cycle. We first show that $feps_0 < 2^{-26}$. The reason is that instead of computing $2 - D'_0[0 : 28]$, we compute $2 - D'_0[0 : 26]$. The absolute error is $-D'_0[27 : 28]$. Hence, $feps_0 < 2^{-26}$, as required. Next, we expand $2 - D'_0 = 2 - (1 + d_0) \cdot (1 - e_0)$. We conclude that

$$\begin{aligned} f_0 &\leq \frac{2^{-26}}{2 - (1 + d_0) \cdot (1 + |e_0|)} \\ &< 2^{-25.99989454}. \end{aligned}$$

Bound on n_1: As in the case of n_0, $neps_1 < 2^{-27}$.

We bound from below N'_0 and F'_0 as follows: $N'_0 = (1 - n_0) \cdot (1 - e_0) \cdot \frac{A'}{B}$. Hence, $N'_0 \geq (1 - n_0) \cdot (1 - |e_0|)$. Also, $F'_0 = (1 - f_0) \cdot (2 - D'_0) \geq (1 - f_0) \cdot (1 - \delta_0)$. We conclude that

$$\begin{aligned} n_1 &< \frac{2^{-27}}{(1 - n_0) \cdot (1 - |e_0|) \cdot (1 - f_0) \cdot (1 - \delta_0)} \\ &< 2^{-26.999789}. \end{aligned}$$

We now combine these bounds to obtain:

$$\pi_1 + \delta_1 \leq 2^{-24.48458} < 2^{-24}.$$

□

B.5.4 Design alternatives

The proposed implementation is not the only way to implement our algorithm. Certain choices were made in order to keep the implementation simple without increasing cost by much. Below we list a few alternatives.

1. If the length of the first operand of the multiplier is wider than $29 + 30$ bits, then cycles 2 and 3 could be executed simultaneously. The reason is that the two multiplications that take place in these cycles have the same second operand (i.e., F'_{-1}). (Compression of D'_0 is not crucial and could be skipped at the expense of increasing the error term f_0.) This implies a reduction in the latency from 9 cycles to 8 cycles.

2. The normalization of the quotient can be postponed till after N'_1 is computed. This modification has the following implications: (i) A' is not computed, and we simply use A instead. (ii) In this version the rounding injection cannot be added in cycle 5 since normalization changes the value of the injection. Hence in cycle 6 an injection (or injection correction) needs to be added when computing RC_1. (iii) The error analysis should be changed to deal with this case (slightly worse bounds will be obtained).

3. We chose to use a multiplier in which both the first operand and the product have a digit in position $[-1]$. This can be avoided by normalizing the operand and product so that the leftmost digit is always non-zero.

4. If the initial reciprocal approximation has a one-sided error, namely $e_0 \geq 0$, then we can also guarantee that $N'_0 < 2$. This means that the the first operand of the multiplier need not support position $[-1]$. In the second pipeline stage in cycle 3, we could compute instead: $mul_{RZ}(N'_{-1}[0:28], F'_{-1}[0:14]) + N'_{-1}[-1] \cdot F'_{-1}[0:14]$.

B.6 An implementation with a half-sized multiplier

In this section we present an implementation of our algorithm that uses a half-sized multiplier (i.e., roughly $p \times p/2$ for a target precision p) and an initial approximation whose precision is roughly $p/4$. To be concrete, we present detailed design for double precision (i.e. $p_d = 53$) that has a latency of 11 cycles.

The microarchitecture is an extension of the microarchitecture introduced in Section D.6.1. The main differences are in the computations related to the last iteration and the back-multiplication. These changes influence circuitry needed for selection, injections, computation of the rounding candidates, and the dewpoint.

B.6.1 Pipeline schedule

Table B.3 lists the schedule of operations that implements our algorithm for double precision based on a half-sized multiplier and a quarter-sized initial approximation. In the error analysis presented in Section D.6.3 we show that a 62×30 rectangular multiplier together with an error bound of $|e_0| \leq 2^{13.74}$ guarantees correctness. We use these parameters in the detailed description below. We now describe the cycles one by one.

cycle 0-2: Same as cycles 0-2 in Section B.5.2.

cycle 3: In the second pipeline stage, a carry-save representation of the the product $N'_0 \leftarrow N'_{-1} \cdot F'_{-1}$ is computed. In the first and third pipeline stages preparations are made for computing the product $D'_1 \leftarrow D'_0 \cdot F'_0$ in the next cycle. These preparations include the compression of D'_0 to binary, the computation of the $3\times$ multiple of D'_0, and the recoding of F'_0.

Note that the computation $F'_0 \leftarrow 2-(D'_0[0:29]+2^{-28}$ is done simply by complementing all the bits in the carry-save representation of $D'_0[0:29]$. The carry-save number representing F'_0 is then directly recoded without having to compress it.

cycle 4: In the second pipeline stage, the product $D'_1 \leftarrow D'_0 \cdot F'_0$ is computed. In the first and third pipeline stages, preparations are made for computing the product $N'_1 \leftarrow N'_0 \cdot F'_0$ in the next cycle. Namely, N'_0 is compressed and the 3× multiple of N'_0 is computed.

cycle 5**:** In the second pipeline stage, the product $N'_1 \leftarrow N'_0 \cdot F'_0$ is computed. In the third pipeline stage, D'_1 is compressed to reduce the error f_1 associated with the computation of F'_1.

cycle 6**:** In cycle 6 preparations are made for computing $N'_2 \leftarrow N'_1 \cdot F'_1$ in the next cycle. These preparations include compressing N'_1, computing the $\times 3$ multiple of N'_1. As for F'_1, we rely here on the assumption that $F'_1 \in (1, 1 + \delta_1]$ (see Section B.7.5). This implies that we only need to compute the lower bits of F'_1 (i.e., bits in positions greater than or equal to $\log_2(1/\delta_1)$). In Section B.6.2, we show that $F'_1[1 : 26]$ is all zeros. Hence, we compute $F'_1[27 : 56]$ simply by complementing the bits of $D'[27 : 56]$. We then proceed by recoding $F'_1[27 : 56]$.

cycle 7**:** In the second pipeline stage, the product $N''_2 \leftarrow N'_1 \cdot F'_1 + \text{INJ}_{rnd}(mode)$ is computed. Our implementation is based on the fact that F'_1 is slightly larger than 1. Therefore we use the cheaper method of computing $N'_1 + N'_1 \cdot (F'_1 - 1)$ instead of $N_1 \cdot F'_1$. (See Section B.7.5.)

In the first pipeline stage, preparations are made for the back-multiplication $B \cdot dewpoint - A'$. The definition of the dewpoint depends on N'_2, which is not ready at this cycle. We therefore use an approximation of the the dewpoint. We denote this approximation by $dewpoint^h$. We derive $dewpoint^h$ from the upper half of N'_1. (See also Section D.5.3 for more details.)

In the third pipeline stage, we compute the two's complement of $dewpoint^h$. We need this number for the next cycle for subtracting $dewpoint^h$.

cycle 8**:** The first part of back-multiplication takes place in the second pipeline stage. We compute $\alpha^H \leftarrow B \cdot dewpoint^h - A'$. Note that, the final decision is based on positions [51 : 105] (see Section D.5.3). Hence a shortened representation of $4 - A'$ suffices. Namely, instead of adding $4 - A'$, we could simply add $2^{50} - A'[51 : 52]$.

In the third pipeline stage, we compute RC_1 by compressing N_2'' and truncating it at position 52.

In the first pipeline stage we compute the difference between the correct dewpoint and the estimate $dewpoint^h$. We denote this difference by $dewpoint^\ell$. Note that a borrow-save representation of $dewpoint^\ell$ suffices since it only needs to be recoded in the first pipeline stage. See Section D.5.3 for more details on the computation of $dewpoint^\ell$. We then recode $dewpoint^\ell$, and the $3\times$ multiple of B is prepared for the next cycle.

cycle 9**:** The second part of the back-multiplication takes place in the second pipeline stage.

cycle 10**:** In the third pipeline stage, (i) a compressed representation of RC_2 is computed by incrementing RC_1, (ii) the z-sign of the back-multiplication is determined from $\alpha^L[51 : 105]$, and (ii) the IEEE rounded quotient is selected from RC_1 and RC_2 according to the rounding mode and the z-sign of α.

B.6.2 Correctness

In this section we prove the correctness of a design based on a rectangular 62 × 30 multiplier and an initial approximation with a relative error $e_0 \leq 2^{-13.74}$. To support double precision ($p_d = 53$), dewpoint rounding requires that Assumption 19 holds. Namely, $0 \leq \rho(N_2') < 2^{-53}$.

The analysis presented in this section is a concrete analysis for double precision. More generally, this analysis applies to the following situation: (i) the accuracy of the initial approximation is roughly $2^{-p/2}$; (ii) the quotient is approximated by N_2' (namely, the algorithm has 2 iterations of quadratical convergence); and (iii) the multiplier is a rectangular multiplier whose dimensions are not smaller than roughly $p_d \times p_d/2$.

Claim 23 *If the relative error in initial approximation satisfies* $|e_0| \leq 2^{-13.74}$*, then correct double precision IEEE rounding is obtained with a rectangular* 62×30*-bit multiplier.*

Proof: All we need to show is that $0 \leq \rho(N_2') < 2^{-53}$. By Theorem 15, $0 \leq \rho(N_2') \leq \pi_2 + \delta_2$. We expand π_2 and δ_2 to obtain:

$$\begin{aligned} \pi_2 &= 1 - (1 - n_2) \cdot \frac{1 - n_0}{1 + d_0} \cdot \frac{1 - n_1}{1 + d_1} \\ \delta_2 &= f_1 + \left(\left(|e_0| + \frac{3}{2} \cdot d_0 \right)^2 + f_0 \right)^2 . \end{aligned} \tag{B.12}$$

To prove $\pi_2 + \delta_2 < 2^{-53}$, we prove upper bounds on the relative error terms (i.e., n_i, f_i, d_i) and substitute these upper bounds in Equation B.12. We bound these relative errors using the absolute error terms $neps_i, deps_i, feps_i$ defined in Definition 6.

We now bound each of the relative error terms:

Bound on d_0: According to Table B.3, D_0' is represented by a carry-save encoded digit string with digits in positions $[0 : 60]$. This digit string holds the *rounded up* value of $D'_{-1} \cdot F'_{-1}$. This implies that the absolute error $deps_0$ associated with the computation of D_0' is in the range $[0, 2 \cdot 2^{-60})$. Since the precise product $D'_{-1} \cdot F'_{-1}$ equals $1 - e_0$, this product is in the range $1 \pm 2^{-13.74}$. It follows that $d_0 < 2^{-59}/(1 - 2^{-13.74}) < 2^{-58.99989}$.

Bound on n_0: As in the case of D_0', N_0' is also represented by a carry-save encoded digit string with digits in positions $[0 : 60]$. This digit string holds the *rounded down* value of $N'_{-1} \cdot F'_{-1}$. This implies that the absolute error $neps_0$ associated with the computation of D_0' is in the range $[0, 2 \cdot 2^{-60})$. The precise product $N'_{-1} \cdot F'_{-1}$ equals $\frac{A'}{B} \cdot (1 - e_0)$, and is therefore at least $(1 - |e_0|)$. We conclude that $n_0 < 2^{-59}/(1 - |e_0|) < 2^{-58.99989}$.

Bound on f_0: We first show that $feps_0 < 2^{-28}$. The reason is that instead of computing $2 - D_0'[0 : 60]$, we compute $2 - D_0'[0 : 29] - 2^{-28}$. The absolute error is $2^{-28} - D_0'[30 : 60]$. Since the tail $0 \leq D_0'[30 : 60] < 2 \cdot 2^{-29}$, we conclude that indeed $0 \leq feps_0 < 2^{-28}$ (recall that $D'[0 : 60]$ is represented by a carry-save encoded digit string). Next, we expand

$2 - D'_0 = 2 - (1 + d_0) \cdot (1 - e_0)$. We conclude that

$$f_0 \leq \frac{2^{-28}}{2 - (1 + d_0) \cdot (1 + |e_0|)}$$
$$< 2^{-27.999894}.$$

Bound on d_1: As in the bound on d_0, $deps_1 < 2^{-59}$. We bound from below the exact product as follows: $D'_0 \cdot F'_0 = (1 - f_0) \cdot D'_0 \cdot (2 - D'_0)$. Now $D'_0 = (1 + d_0)(1 - e_0)$, hence $D'_0 \in [1 - |e_0|, 1 + \delta_0)$. It follows that $D'_0 \cdot F'_0 \geq (1 - f_0) \cdot (1 - \delta_0^2) \geq 1 - \delta_1$. We conclude that

$$d_1 < \frac{2^{-59}}{1 - \delta_1}$$
$$< 2^{-58.999999}.$$

Bound on n_1: Again, we use $neps_1 < 2^{-59}$. We bound from below N'_0 and F'_0 as follows: $N'_0 = (1 - n_0) \cdot (1 - e_0) \cdot \frac{A'}{B}$. Hence, $N'_0 \geq (1 - n_0) \cdot (1 - |e_0|)$. Also, $F'_0 = (1 - f_0) \cdot (2 - D'_0) \geq (1 - f_0) \cdot (1 - \delta_0)$. We conclude that

$$n_1 < \frac{2^{-59}}{(1 - n_0) \cdot (1 - |e_0|) \cdot (1 - f_0) \cdot (1 - \delta_0)}$$
$$< 2^{-58.999789}.$$

Bound on f_1: Assume that $D'_1 \in [1 - \delta_1, 1)$. Note that according to Table B.3, when computing F'_1 (during cycle 6) we have a binary non-redundant representation of D'_1 (from cycle 5). Since

$$\delta_1 = \delta_0^2 + f_0$$
$$< 2^{-26.716654},$$

It follows that the binary representation of D'_1 satisfies $D'_1[0:26] = 1 - 2^{-26}$. We conclude that

$$\begin{aligned} 2 - D'[0:60] &= \left(1 + (1 - 2^{-26}) + 2^{-26}\right) - (D'_1[0:26] + D'_1[27:60]) \\ &= 1 + 2^{-26} - D'_1[27:60]. \end{aligned}$$

Instead of computing $2 - D'_1[0:60]$, we compute (in cycle 6) $1 + 2^{-26} - (D'_1[27:56] + 2^{-56})$ (note that the 1 does not appear explicitly since it does not effect the actual computation). It follows that

$$\begin{aligned} feps_1 &= \left(1 + 2^{-26} - D'_1[27:60]\right) - \left(1 + 2^{-26} - (D'_1[27:56] + 2^{-56})\right) \\ &= 2^{-56} - D'[57:60] \\ &\leq 2^{-56}. \end{aligned}$$

Since we assumed that $D'_1 < 1$, it follows that $2 - D'_1 > 1$. Hence $f_1 \leq feps_1 \leq 2^{-56}$.

If $D'_1 > 1$, then, by Equation B.3 in the proof of Theorem 15, we know that

$$\begin{aligned} \rho(N'_1) &\leq 1 - \left(\frac{1 - n_0}{1 + d_0}\right) \cdot \left(\frac{1 - n_1}{1 + d_1}\right) \\ &< 2^{-56.999894}. \end{aligned}$$

However, by Assumption 14 (precise multiplication by 1), it follows that $N'_1 = N'_2$. Hence, if $D'_1 > 1$, then we obtain the desired relative error.

Bound on n_2: As before, $neps_2 < 2^{-59}$.

The precise product $N'_1 \cdot F'_1$ is bounded as follows. Saturated rounding implies that $F'_1 \geq 1$.

Theorem 15 implies that $N_1' \geq \frac{A'}{B} \cdot (1 - \pi_1 - \delta_1)$. We conclude that

$$\begin{aligned} n_2 &\leq \frac{2^{-59}}{(1 - \pi_1 - \delta_1)} \\ &< 2^{-57.9999999}. \end{aligned}$$

We now combine these bounds to obtain:

$$\pi_2 + \delta_2 \leq 2^{-53.05992} < 2^{-53}.$$

□

We point out that Claim 23 is tight in the sense that if we decrease any of the three parameters: initial approximation error, long operand length, or short operand length, then we can no longer prove that $\rho(N_2') < 2^{-53}$.

B.7 Hardware Optimizations

B.7.1 Redundant Feedback & Partial Compression

Addition trees in multipliers are not amenable to pipelining, so short clock cycles are not achievable with reasonable cost if the addition tree has too many rows. Booth radix 8 recoding reduces the number of rows in an addition tree from n to $\lceil \frac{n+1}{3} \rceil$.

Booth radix-8 multipliers are usually implemented using a 3-stage pipeline (i.e. (i) precompute the 3× multiple of the first operand of the multiplier and recode the second operand, (ii) addition tree, and (iii) final carry-propagate addition). Goldschmidt's algorithm performs only 2 multiplications per iteration. Hence running Goldschmidt's algorithm on a 3-stage pipeline creates un-utilized cycles (i.e. "bubbles") in the pipeline. The Athlon™ processor (in hardware) [28] and Itanium™ processor (in software) [5] allow other multiplications to be executed during such bubbles.

Following Seidel [35], we manage to get rid of all of these bubbles but one by allowing redundant operands. Conceptually, we skip the third pipeline stage and reduce the pipeline to a two-stage pipeline for all but the last iteration of the algorithm. This is obtained by feeding back results represented in carry-save format. This design is not symmetric in the sense that the first operand and second operand of the multiplier are processed differently during the first pipeline stage. During the first pipeline stage, operands represented as carry-save number are processed as follows:

1. The first operand is compressed and its $3\times$ multiple is computed. This requires two adders. To save hardware, we suggest "borrowing" the adder from the third pipeline stage for the purpose of compressing the first operand. This explains why we listed the compression of the first operand as an operation that takes place in the third pipeline stage in Tables B.2 and B.3.

2. The second operand can be partially compressed from carry-save representation before being fed to the Booth recoder.

B.7.2 Normalization by checking $(A \geq B)$

Suppose that the operands satisfy $|A|, |B| \in [1, 2)$. We compare the operands A and B. If $|A| < |B|$, then we set $|A'| \leftarrow 2 \cdot |A|$; otherwise $|A'| \leftarrow |A|$. This ensures that $|A'|/|B| \in [1, 2)$ is the value of the normalized quotient. This normalization also slightly improves the bound on the error.

B.7.3 Directed rounding of carry-save numbers

Our algorithm has to deal with rounding in two situations. The first situation is when directed rounding is used in intermediate computations. In this situation, the intermediate product is represented by a carry-save encoded digit string. We wish to apply directed rounding without incurring a significant delay. Directed rounding of carry-save number is described in this section. The second situation is when we wish to determine the correct IEEE rounded quotient; we deal with IEEE rounding in Section B.4.

We propose an implementation of Algorithm 2 in which intermediate results are represented by carry-save encoded digit strings. Since Algorithm 2 uses directed roundings for all intermediate computations, we need to explain how directed rounding is applied to carry-save numbers.

Notation. Consider three binary vectors x, y, z. We denote 3:2-addition by $FA(x, y, z)$. Namely, $FA(x, y, z)$ means that the three binary vectors x, y, z are fed to a line of full-adders. The output of $FA(x, y, z)$ is two binary vectors s and c (i.e., a carry-save number) that satisfy $|x| + |y| + |z| = |s| + |c|$.

By *truncating* a carry-save encoded digit string $\sigma[i : j]$ after position q we simply mean chopping off the tail and leaving only $\sigma[i : q]$. We denote truncation of σ after position q by $\lfloor \sigma \rfloor_q$.

We often regard a carry-save encoded digit string σ also as two binary vectors σ' and σ''. Hence if σ is a carry-save encoded digit string and x is a binary vector, then $FA(\sigma, x)$ simply means $FA(\sigma', \sigma'', x)$.

The following lemma deals with rounding up of a carry-save number. It shows that 3:2-addition followed by truncation after position q can be used to implement approximate rounding up. The 3:2-addition adds a constant, called the injection, that depends only on the rounding position and the length of the number.

Lemma 24 *Let $\sigma[0 : t]$ denote a carry-save encoded digit string. Let $\text{INJCS}_{RU}(q, t) \in [2^{-q} + 2^{-(q+1)}, 2 \cdot (2^{-q} - 2^{-t})]$ and assume that $\text{INJCS}_{RU}(q, t)$ is represented by a binary string $I[q : t]$. Then,*

$$|\sigma[0 : t]| \leq |\lfloor FA(\sigma[0 : t], I[q : t]) \rfloor_q| \leq |\sigma[0 : q]| + 2^{-q+1}.$$

Proof: We first show that

$$|\sigma[0 : t]| \leq |\lfloor FA(\sigma[0 : t], I[q : t]) \rfloor_q|. \tag{B.13}$$

From the range of $\text{INJCS}_{RU}(q,t)$, it follows that $I[q] = 1$ and $I[q+1] = 1$. Hence,

$$|\lfloor FA(\sigma[0:t], I[q:t])\rfloor_q| = |\sigma[0:q]| + 2^{-q} + C \cdot 2^{-q}, \tag{B.14}$$

where C denotes the carry bit generated by adding $I[q+1] + \sigma[q+1]$. We consider two cases: (i) $C = 0$: in this case $\sigma[q+1] = 0$, and hence, $|\sigma[q+1:t]| < 2^{-q}$. It follows that Equation D.9 holds in this case. (ii) $C = 1$: in this case $\sigma[q+1:t] \leq 2^{-(q-1)}$, and hence Equation D.9 also holds.

We now show that

$$|\lfloor FA(\sigma[0:t], I[q:t])\rfloor_q| \leq |\sigma[0:q]| + 2^{-q+1}. \tag{B.15}$$

This follows directly from Equation D.10 since $|\sigma[0:q]| + 2^{-q} + C \cdot 2^{-q} \leq |\sigma[0:q]| + 2^{-q+1}$. □

We present in Lemma 43 the widest possible interval for $\text{INJCS}_{RU}(q,t)$. It is probably most beneficial to use a value with the smallest number of ones, namely, $\text{INJCS}_{RU}(q,t) = 2^{-q} + 2^{-(q+1)}$.

B.7.4 Implementation of Saturated Rounding

The definition of saturated rounding from Assumption 2 requires that $F'_i \geq 1$ for $i > 0$. Suppose that $D'_i \geq 1$, then $2 - D'_i \leq 1$, and hence F'_i should equal 1. Moreover, Assumption 14 requires that every multiplication by 1 is precise.

We propose the following implementation. Multiplication takes place in the second pipeline stage. During a multiplication by F'_i that is part of iteration $i > 0$, the algorithm checks if $D'_i \geq 1$. If $D'_i \geq 1$, then we simply de-activate the clock enable signal of the register that is supposed to store the result of the multiplication. The effect of ignoring the multiplication means that the multiplicand is not changed, i.e., precise multiplication by 1 is obtained.

We test whether $D'_i \geq 1$ as follows. Recall that D'_i is represented in carry-save format. We subtract 1 from the D'_i using a signed 3:2-adder (i.e., PPM-add [26]) to obtain a borrow-save number. The borrow-save number is fed to a subtracter, and the decision is made according to the value of the sign bit of the difference.

B.7.5 Half-size multiplication in the last iteration

The precision in the last iteration must be at least p_d to obtain a sufficiently close approximation of the quotient. This presumably implies that the length of the second operand (i.e. the multiplier) should be roughly p_d. In Section B.6 we present an implementation that only uses a single pass through a half-sized multiplier. This implementation is based on the error analysis and, in particular, on Corollary 17.

For the sake of concreteness we consider double precision ($p_d = 53$). The last iteration requires computing the product $N_2' = N_1' \cdot F_1'$. The length of F_1' should be bigger than $p_d = 53$ by a few bits; otherwise the relative error f_1' would be too large. How can we avoid having to multiply N_1' by the full length of F_1'?

Rewrite N_2' as follows:

$$\begin{aligned} N_2' &= N_1' \cdot (1 + (F_1' - 1)) && \text{(B.16)} \\ &= N_1' + N_1' \cdot (F_1' - 1). && \text{(B.17)} \end{aligned}$$

By Corollary 17, $(F_1' - 1) \in [0, \delta_1]$. Hence, we may ignore bit positions $[1 : j]$ where $j = \lfloor \log_2 \frac{1}{\delta_1} \rfloor$. We conclude that $N_2' = N_1' + N_1' \cdot F_1'[j : t]$, where $t = \lceil \log_2 \frac{1}{\delta_1} \rceil$. Our error analysis shows that $t - j$ is roughly $p_d/2$, as required.

The addition of N_1' to the product $N_1' \cdot F_1'[j : t]$ can be done by using one of the extra rows in the adder tree. The second extra row is used for feeding the rounding injection $\text{INJ}_{rnd}(\mathit{mode})$. Note that RZ is used for the computation of N_2', hence no carry-save injection is required.

B.7.6 Optimized implementation of Dewpoint Rounding and Back multiplication

In Section B.4.2 we showed how the IEEE rounded quotient is chosen to be either RC_1 or RC_2 based on the z-sign (positive, negative, zero) of $\mathit{dewpoint} \cdot |B| - |A'|$.

For the consideration of negative numbers and signs, we need to consider two's complement representations. In particular we consider a representation called two's complement carry-save representation [7]. In this representation, a number is represented by two binary vectors, each of which is a two's complement number, namely, the most significant position has a negative weight. Our goal in the back multiplication is to compute the sign of $\alpha = |B| \cdot \mathit{dewpoint} - |A'|$, we also want to know if α is precisely zero to determine the condition for rounding mode RI. The z-sign of a number x is either positive, negative, or zero depending on the result of comparing x with zero. Our goal is to compute $z\text{-sign}(\alpha)$.

We now discuss two optimization techniques. The first technique is used to simplify the computation of $z\text{-sign}(\alpha)$. The second technique is used in double precision to split the back-multiplication into two passes through the half-sized multiplier, where in the first pass an estimate of the dewpoint is used. The estimated dewpoint is based on N_1' rather than N_2'. This means that we begin the first pass of the back-multiplication before N_2' is ready.

Computing z-sign(α). We first prove the following claim.

Claim 25

$$-2 \cdot 2^{-p} \leq |B| \cdot \mathit{dewpoint} - |A'| \leq 2 \cdot 2^{-p}.$$

Proof: By Assumption 19, $0 \leq \rho(N_2') < 2^{-p}$. Recall that

$$\mathit{dewpoint} = \lfloor N_2' + \text{INJ}_{rnd}(\mathit{mode}) \rfloor_{p-1} + C_{dew}(\mathit{mode}).$$

It follows that

$$\begin{aligned} dewpoint &\leq N_2' + \text{INJ}_{rnd}(mode) + C_{dew}(mode) \\ &\leq N_2' + 2^{-p+1} \\ &\leq \frac{A'}{B} + 2^{-p+1}. \end{aligned}$$

The second line follows from the definition of the rounding injection $\text{INJ}_{rnd}(mode)$ and the dewpoint displacement constant $C_{dew}(mode)$. The third line follows from $0 \leq \rho(N_2')$.

Since $\rho(N_2') < 2^{-p}$ and $A'/B < 2$, the other part of the claim follows if we show that $dewpoint \geq N_2'$. We consider each of the three rounding modes.

RZ: Let x and x' denote two successive representable significands that sandwich N_2', namely, $N_2' \in [x, x')$ where $x' = x + 2^{-p+1}$. In RZ, $\text{INJ}_{rnd}(RZ) = 0$ and $C_{dew}(RZ) = 2^{-p+1}$. Hence: $\lfloor N_2' + \text{INJ}_{rnd}(mode) \rfloor_{p-1} = x$ and $dewpoint = x'$.

RI: Assume here that $N_2' \in (x, x']$ (i.e., the equality can hold in x' instead of x). In RI, $\text{INJ}_{rnd}(RI) = 2^{-p+1} - 2^{-t}$ and $C_{dew}(RNU) = 0$. Hence $\lfloor N_2' + \text{INJ}_{rnd}(mode) \rfloor_{p-1} = x'$ and $dewpoint \geq x'$.

RNU: Assume here that $N_2' \in [x + 2^{-p}, x' + 2^{-p})$ (i.e., we sandwich N_2' between two odd multiples of 2^{-p}. In RNU, $\text{INJ}_{rnd}(RNU) = 2^{-p}$ and $C_{dew}(RNU) = 2^{-p}$. Hence $\lfloor N_2' + \text{INJ}_{rnd}(mode) \rfloor_{p-1} = x'$ and $dewpoint = x' + 2^{-p}$.

In all three rounding modes, $dewpoint \geq N_2'$, and hence the claim follows. □

Let $a[-2 : t]$ denote a two's complement non-redundant representation of α, namely:

$$\alpha = -a_{-2} \cdot 2^2 + \sum_{i=-1}^{t} a_i \cdot 2^{-i}.$$

By Claim 44 it follows that $-a_{-2} \cdot 2^2 + \sum_{i=-1}^{p-2} a_i \cdot 2^{-i} \in \{-2^{p-2}, 0\}$. Hence, the bits in positions $[-2 : p-2]$ are either all zeros or all ones. We conclude that

$$\alpha = 0 \quad \text{iff} \quad \text{OR}(a_{p-2}, \ldots, a_t) = 0, \tag{B.18}$$

$$\alpha < 0 \quad \text{iff} \quad a_{p-2} = 1. \tag{B.19}$$

Note that the multiplier computes a two's complement carry save representation of α. For the purpose of computing z-sign(α), we conclude that the carry-save digits of α in positions $[p-2 : t]$ suffice. This last remark also implies that we need not compute $4 - A'$ since it suffices to use $2^{-(p-3)} - A'[p-2 : p-1]$.

Back-multiplication with an estimated dewpoint. In Section B.6, back-multiplication is implemented by two passes through the half-sized multiplier. An estimate of the dewpoint is used in the first multiplication. We denote this estimate by $dewpoint^h$. The estimate $dewpoint^h$ is computed by truncating N_1' to fit the length of the second operand of the multiplier. In the first pass we compute

$$\alpha^H \leftarrow B \cdot dewpoint^h + (4 - A'). \tag{B.20}$$

In the second pass we correct the computation. The difference between the dewpoint and the estimated dewpoint is denoted by $dewpoint^\ell$. Formally,

$$\begin{aligned} dewpoint^\ell &= dewpoint - dewpoint^h \\ &= RC_1 + C_{dew}(mode) - dewpoint^h. \end{aligned}$$

In the second pass we compute

$$\alpha^L \leftarrow B \cdot dewpoint^\ell + \alpha^H. \tag{B.21}$$

Note that Equations B.20 and B.21 imply that $\alpha^L = \alpha$. Based on Equations B.18 and B.19, we are only interested in $\alpha^L[p-2:t]$.

We now compute which digit positions are required for $dewpoint^\ell$. To be concrete we focus on double precision as analyzed in Section B.6. The estimate $dewpoint^h$ is simply the truncation of the non-redundant representation of N_1'. Since the multiplier's second operand length is 30, we set

$$dewpoint^h \leftarrow N_1'[0:29].$$

Claim 26 *Under the premises of Claim 23,* $0 \leq dewpoint^\ell \leq 2^{-25.447}$.

Proof: The rounding candidate RC_1 equals the truncation of $N_2'[0:60] + \text{INJ}_{rnd}(mode)$ at position 52 (note that the carry bit for position [52] resulting from compressing the carry-save representation of N_2' is computed). Note that $N_2' = (1-n_2) \cdot F_1' \cdot N_1'$. Since $F_1' \leq 1 + \delta_1$, it follows that $N_2' \leq (1+\delta_1) \cdot N_1'$. Hence,

$$\begin{aligned} dewpoint^\ell &\leq (1+\delta_1) \cdot N_1'[0:60]| + \text{INJ}_{rnd}(mode) + C_{dew}(mode) - N'[0:29] \\ &\leq (1+\delta_1) \cdot N_1'[30:60] + \delta_1 \cdot N_1'[0:29] + 2^{-52}. \end{aligned} \tag{B.22}$$

The last equation follows from $\text{INJ}_{rnd}(mode) + C_{dew}(mode) \leq 2^{-p+1}$. We bound $N_1'[0:29] \leq 2$, $N_1'[30:60] \leq 2^{-29}$, and $\delta_1 \leq 2^{-26.716654}$ (from the proof of Claim 23) to obtain the upper on $dewpoint^\ell$. The lower bound follows simply because $RC_1 \geq dewpoint^h$. □

Claim 26 implies that if $dewpoint^\ell$ is represented as a carry-save number, then it suffices to represent $dewpoint^\ell$ using digits in positions [26 : 53]. This means that $dewpoint^\ell$ easily fits as the second operand of the multiplier (if shifted by 26 positions). Finally, since we are only interested in positions to the left of $p_d - 2$, we conclude that Equations B.20 and B.21 can be implemented as follows:

$$\alpha^H[0:81] \leftarrow B[0:52]\cdot \mathit{dewpoint}^h[0:29] + (2^{50} - A'[51:52])$$
$$\alpha^L[26:105] \leftarrow B[0:52]\cdot \mathit{dewpoint}^\ell[26:53] + \alpha^H[51:81].$$

Note that this means that in the second pass we need to also shift $\alpha^H[51:81]$ by 26 positions before we inject it to the adder tree.

Computation of $\mathit{dewpoint}^\ell[26:53]$ The implementation of the dewpoint computation is depicted in figure B.3. We explain the details of the implementation below. The operand $\mathit{dewpoint}^\ell[26:53]$ is required in cycle 8 for the double precision implementation (see table B.3). A Booth recoded representation of the value $\mathit{dewpoint}^\ell[26:53]$ needs to be computed from the carry-save representation of $N_2''[0:60]$. The computation is based on the equation:

$$\mathit{dewpoint}^\ell[0:53] \;=\; N_2''[0:52] + \mathit{carry}_{52}(N_2''[53:60]) + C_{\mathit{dew}}(\mathit{mode}) - \mathit{dewpoint}^h[0:29].$$

By Claim 26, $0 \le \mathit{dewpoint}^\ell < 2^{-25}$. Hence, it suffices to compute bit positions $[26:53]$ of the binary representation of $\mathit{dewpoint}^\ell$. However, computing the binary representation is slow and is not needed for obtaining a Booth radix-8 recoding of $\mathit{dewpoint}^\ell$.

We precompute (part of) $2 - \mathit{dewpoint}^h[0:29]$ in cycle 7. Consider a carry-save number $RCS[-1:53]$ defined by

$$RCS[-1:53] \;\triangleq\; N_2''[0:52] + \mathit{carry}_{52}(N_2''[53:60]) + C_{\mathit{dew}}(\mathit{mode}) + (2 - \mathit{dewpoint}^h[0:29]).$$

Obviously, the value represented by $RCS[-1:53]$ equals $2 + \mathit{dewpoint}^\ell$.

Our goal is to compute a Booth radix-8 recoding of $\mathit{dewpoint}^\ell$. We denote a Booth radix-8 recoding of $\mathit{dewpoint}^\ell$ by $BD_{\mathit{dew}}[-1:53]$, where $BD_{\mathit{dew}}[3i+2] \in [-4,4]$ is a Booth radix-8 digit, for every $i = -1, 0, \ldots, 17$. First, we observe that (i) $BD_{\mathit{dew}}[-1:23] = 0$, namely the Booth digits in positions $[-1:23]$ are all zeros, and (ii) the most significand non-zero Booth digit in $BD_{dew}[-1 : 53]$ is positive. This follows from the fact that $dewpoint^{\ell} \in [0, 2^{-25})$ and since the fraction range of a Booth radix 8 number is $(-4/7, 4/7)$. Hence, we are left with the task of computing $BD_{dew}[26 : 53]$.

We apply the following method of [35] for obtaining a minimally redundant format in which each block of 3 positions represents a value in the range $[-4, 4]$. First we feed the carry-save number RCS to a line of 3-bit adders. The output of the line of 3-bit adders for positions $[3i : 3i + 2]$ consists of 4 bits: one bit for position $[3i]$ has weight $2^{-3i} = 4 \cdot 2^{-(3i+2)}$, one bit for position $[3i+1]$ has weight $2^{-3i-1} = 2 \cdot 2^{-(3i+2)}$, and two bits for position $[3i+2]$ each have weight $2^{-(3i+2)}$. It follows that this 4-bit string represents a digit in the range $[0, 8]$ whose weight is $2^{-(3i+2)}$. To allow for interpretation of each string as a digit in the range $[-4, 4]$, we add to RCS a constant. Loosely speaking, the constant increases each block $[3i : 3i + 2]$ by $4 \cdot 2^{-(3i-2)}$. This implies that subtracting 4 from each digit is in effect a subtraction of the added constant. By subtracting 4 from each digit, we translate digits in $[0, 8]$ to Booth radix-8 digits in $[-4, 4]$. We denote the Booth digit correspoding to block $[3i : 3i + 2]$ by $BD[3i + 2]$.

Based on this discussion our circuit does not compute RCS. Instead, the algorithm computes (part of) a carry-save number $R[-1 : 53]$ that is defined by

$$R[-1 : 53] \triangleq RCS[-1 : 53] + 2^3 + 2^0 + \cdots + 2^{-24} + 2^{-27} + \cdots + 2^{-51}.$$

The carry-save number $R[24 : 53]$ is fed to a line of 3-bit adders that outputs (part of) the sum-string $S_3[-1 : 53]$ and (part of) the carry-string $C_3[-2 : 52]$. Note that the carry-string C_3 has bits only in position $3i + 2$, for each block $[3i : 3i + 2]$.

We define the Booth digit $BD[3i + 2]$ by:

$$BD[3i + 2] \triangleq S_3[3i : 3i + 2] + C_3[3i + 2] - 4.$$

Note that the above discussion in which a constant is added and then subtracted from RCS implies that $RCS[-1:53] = BD[-1:53]$.

Finally, we define $BD'[26]$ as the Booth digit in position [26] of $R' = R - 2^{-24}$. Of course, $BD'[26]$ is not uniquely defined, so we describe how we compute it. The Booth digit $BD'[26]$ is computed by not adding the constant 2^{-24} in the left most 3-bit adder (see Fig. B.3). We then use a 3-bit adder to compute the sum bits $S'_3[24:26]$, and define

$$BD'[26] \triangleq S'_3[24:26] + C_3[26] - 4.$$

Claim 27 *Let* $B[24:29]$ *denote the binary representation of* $N''_2[25:30] + 2^{-24} - dewpoint^h[25:29]$. *Then,*

$$dewpoint^\ell = BD_{dew}[26:53] = \begin{cases} BD[26:53] & \text{if } B[24:25] = 00 \\ BD'[26] \circ BD[29:53] & \text{otherwise.} \end{cases}$$

Proof: To shorten notation we define α, β, and γ as follows:

$$\alpha \triangleq N''_2[-1:24] + 2 - 2^{-24} - dewpoint^h[0:24]$$
$$\beta \triangleq N''_2[25:30] + 2^{-24} - dewpoint^h[25:29]$$
$$\gamma \triangleq N''_2[31:52] + carry_{52}(N''_2[53:60]) + C_{dew}(mode).$$

Note that $\alpha + \beta + \gamma = 2 + dewpoint^\ell$ and that $\beta + \gamma = BD[26:53]$.

1. $B[24:25] = 00$. In this case $\beta \leq 2^{-25} - 2^{-29}$. Since $\gamma \leq 2^{-29} + 2^{-53}$, it follows that $\beta + \gamma < 2^{-24}$. Now α is a multiple of 2^{-24}, and $2 + 2^{-25} > \alpha + \beta + \gamma \geq 2$. Hence $\alpha = 2$. It follows that $dewpoint^\ell = \beta + \gamma = BD[26:53]$, as required.

2. $B[24:25] = 01$ or $B[24:25] = 10$. In this case $2^{-25} \leq \beta \leq 2^{-24} + 2^{-25}$. It follows that $2^{-25} \leq \beta + \gamma < 2 \cdot 2^{-24}$. If $\alpha \geq 2$, then $\alpha + \beta + \gamma \geq 2 + 2^{-25}$, a contradiction. If $\alpha \leq 2 - 2 \cdot 2^{-24}$, then $\alpha + \beta + \gamma < 2$, a contradiction. Hence, $\alpha = 2 - 2^{-24}$. We conclude that

$\mathit{dewpoint}^{\ell} = \alpha+\beta+\gamma-2 = \beta+\gamma-2^{-24}$. Note that $BD'[26] \circ BD[29:53] = \alpha+\beta-2^{-24}$, as required in this case. (There is a subtle issue which we should address here as well as in the previous case. Namely, show that $C_3'[23] = 0$. This follows from $\mathit{dewpoint}^{\ell} \in [0, 2^{-25})$. If $C_3'[23] = 1$, then $\mathit{dewpoint}^{\ell} \geq 2^{-23} \cdot (1 - 4/7) > 2^{-25}$, a contradiction.)

3. $B[24:25] = 11$. In this case $2^{-24} + 2^{-25} \leq \beta < 2^{-23} - 2^{-28}$. Therefore, $2^{-24} + 2^{-25} \leq \beta + \gamma < 2^{-23}$. If $\alpha \geq 2 - 2^{-24}$ then $\alpha + \beta + \gamma \geq 2 + 2^{-25}$, a contradiction. If $\alpha \leq 2 - 2^{-23}$ then $\alpha + \beta + \gamma < 2$, a contradiction. Hence, this case cannot occur.

□

Finally, note the same hardware can also be used for single precision implementation by unifying and aligning the rounding position for both cases with a 3-bit right shift in the single precision case. The implementation depicted in figure B.3 and figure B.2 presents such an integration for single and double precision.

B.8 Pipelining options

In Section B.6 we presented a micro-architecture that implements our algorithm. The core of this three stage micro-architecture is a half-sized multiplier.

In Tables B.2-B.3 we showed that single precision division can be computed in 9 clock cycles and double precision division can be computed in 11 clock cycles (this includes one cycle for reciprocal approximation). It is easy to see that one can schedule a new independent division every 6 cycles in single precision and every 8 cycles in double precision. In Table B.1, we refer to this micro-architecture as "proposed design (1/2)".

We now discuss how to reduce the restart-times between independent divisions; this requires additional hardware to handle the additional amount of work.

In part (A) of Figure B.4 we present a time-space diagram for double precision division (i.e., $p_d = 53$) based on the schedule defined in Table B.3. The x-axis refers to clock cycles and the

y-axis refers to the four pipeline stages. The squares of the diagram are labeled by the smallest hardware that can execute the corresponding task: $A_{1/4}$ denotes reciprocal approximation that returns roughly $p_d/4$ bits of precision (as well as a comparator), $M^i_{1/2}$ denotes the ith pipeline stage of a half-sized multiplier, and $M^i_{1/4}$ denotes the ith pipeline stage of a quarter-sized multiplier.

Reduced restart-times can be obtained by using more than one multiplier. We propose three designs nicknamed Proposed Designs (1), (2), and (3). Note that a rough cost estimation shows that these designs require about 1×, 2×, and 3× the hardware cost of a conventional multiplicative division implementation (whose core is full-sized multiplier).

Proposed Design (1) Proposed design (1) is built of two copies of half-sized multipliers. Part (B) of Figure B.4 depicts the assignment of tasks to the two multipliers. We refer to the first multiplier by M1 and to the second multiplier by M2. The restart-time of this design is only 4 cycles since M1 and M2 are engaged in the computation for four cycles and three cycles, respectively.

Proposed Design (2) Proposed design (2) is built of three copies of half-sized multipliers and one copy of a quarter-sized multiplier. Part (C) of Figure B.4 depicts the assignment of tasks to the four multipliers. Here we refer to the quarter-sized multiplier by M1 and to the three half-sized multipliers by M2, M3, and M4. The restart-time of this design is only 2 cycles since each multiplier is engaged in the computation for at most two cycles.

Proposed Design (3) Proposed design (3) is built of five copies of a half-sized multiplier and two copies of a quarter-sized multiplier. Part (D) of Figure B.4 depicts the assignment of tasks to the four multipliers. Here we refer to the quarter-sized multipliers by M1-M2 and to the five half-sized multipliers by M3-M7. This design is fully pipelined, and allows for a new independent division every cycle.

The latencies and restart times of the four proposed designs are listed in Table B.1. These designs provide trade-offs between hardware cost and division throughput. The cost ranges from roughly one half up to three times the cost of a regular multiplicative division implementation. The throughput for double precision ranges from one division every 8 cycles up to the fully pipelined implementation that allows to start a new division every cycle.

Similar reductions in restart-times can be applied to the micro-architecture presented in Section D.6. We focus on the situation in which a roughly $p_d \times (p_d/2)$ multiplier is used both for double-precision (i.e., $p_d = 53$) and single precision (i.e., $p_s = 24$). When performing single precision division, the multiplier is a full-sized multiplier. (If fact, the multiplier is two full-sized multipliers.) By applying the same method to the schedule given in Table B.2, the restart-times reported in Table B.1 are obtained.

cycle	op 1st stage	op 2nd stage	op 3rd stage

1	$F'_{-1}[0:14] \leftarrow \frac{1-e_0}{B}$
	$D'_{-1}[0:28] \leftarrow B$
	recode(F'_{-1})
	prepare($3D'_{-1}$)
			If $A < B$ then
			$A'[-1:23] \leftarrow 2 \cdot A[0:23]$,
			else
			$A'[-1:23] \leftarrow A[0:23]$.

2	$N'_{-1}[-1:28] \leftarrow A'[-1:23]$ prepare($3N'_{-1}$)	$D'_0[0:28] \leftarrow mul_{RI}(D'_{-1}[0:28], F'_{-1}[0:14])$	
3		$N'_0[-1:28] \leftarrow mul_{RZ}(N'_{-1}[-1:28], F'_{-1}[0:14])$	compress(D'_0)

4	$F'_0[0:26] \leftarrow 2 - D'_0[0:26]$
	recode($F'_0[0:26]$)
	prepare($3N'_0[0:28]$)
			compress(N'_0)

5		$N''_1[0:28] \leftarrow \quad mul_{RZ}(N'_0[-1:28], F'_0[0:26])$ $+\text{INJ}_{rnd}(mode)$	

6	$dewpoint[0:24] \leftarrow \quad N''_1[0:23] + C_{dew}(mode)$
	$+ \, carry_{23}(N''_1[24:28])$
	recode($dewpoint[0:24]$)
	prepare($3 \cdot B$)
			$RC_1 \leftarrow \lfloor compress(N''_1[0:28]) \rfloor_{23}$

7		$\alpha[0:47] \leftarrow \quad B[0:23] \cdot dewpoint[0:24]$ $+(2^{-21} - A'[22:23])$	
8			$RC_2 \leftarrow RC_1 + 2^{-23}$ compare ($\alpha[22:47] \overset{<}{\underset{>}{=}} 0$) select between RC_1 & RC_2.

Table B.2: Schedule of operations of our FP division implementation with IEEE rounding for single precision operands

cycle	op 1st stage	op 2nd stage	op 3rd stage

1	$F'_{-1}[0:14] \leftarrow \frac{1-e_0}{B}$
	$D'_{-1}[0:60] \leftarrow B$
	recode(F'_{-1})
	prepare($3D'_{-1}$)
			If $A < B$ then $A'[-1:52] \leftarrow 2 \cdot A[0:52]$, else $A'[-1:52] \leftarrow A[0:23]$.

2	$N'_{-1}[-1:60] \leftarrow A'[-1:52]$ prepare($3N'_{-1}$)	$D'_0[0:60] \leftarrow mul_{RI}(D'_{-1}[0:60], F'_{-1}[0:14])$	

3	$F'_0[0:29] \leftarrow 2 - (D'_0[0:29] + 2 \cdot 2^{-29})$
	recode(F'_0)
	prepare($3D'_0$)
		$N'_0[-1:60] \leftarrow mul_{RZ}(N'_{-1}[-1:60], F'_{-1}[0:14])$	compress(D'_0)

4	prepare($3N'_0$)	$D'_1[0:60] \leftarrow mul_{RI}(D'_0[0:60], F'_0[0:29])$	compress(N'_0)
5		$N'_1[0:60] \leftarrow mul_{RZ}(N'_0[-1:60], F'_0[0:29])$	compress(D'_1)

6	$F'_1[27:56] \leftarrow 2^{-26} - (D'_1[27:56] + 2^{-56})$
	recode($F'_1[27:56]$)
	prepare($3N'_1$)
			compress(N'_1)

7	$dewpoint^h[0:29] \leftarrow N'_1[0:29]$
	recode($dewpoint^h[0:29]$)
	prepare($3 \cdot B$)
		$N''_2[0:60] \leftarrow mul_{RZ}(N'_1[0:60], F'_1[27:56])$
		$+ N'_1[0:60] + \text{INJ}_{rnd}(mode)$
			prepare($2 - dewpoint^h$)

8	$dewpoint^\ell[26:53] \leftarrow N''_2[26:52]$ $+carry_{52}(N''_2[53:60]) + C_{dew}(mode)$ $-dewpoint^h[0:29]$ recode($dewpoint^\ell[25:53]$) prepare($3 \cdot B$)	$\alpha^H[0:81] \leftarrow B[0:52] \cdot dewpoint^h[0:29]$ $+(4 - A'[-1:52])$	$RC_1 \leftarrow \lfloor compress(N''_2[0:60]) \rfloor_{52}$
9		$\alpha^L[25:105] \leftarrow B[0:52] \cdot dewpoint^\ell[26:53]$ $+\alpha^H[51:81]$	
10			$RC_2 \leftarrow RC_2 + 2^{-52}$ test($\alpha^L[51:105] \stackrel{<}{\underset{>}{=}} 0$) select between RC_1 and RC_2.

Table B.3: Schedule of operations of the proposed FP division implementation with IEEE rounding for double precision operands

cycle	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	S0	S1	S2	S3	S4	S5	S6	S7	S8	S9
0	1	1	X	X	X	X	X	X	X	X	X	1	X	X	X	X	X	X	X	X	X
1	1	X	X	1	1	1	1	X	X	X	X	0	0	X	X	1	0	0	0	X	X
2	1	X	X	0	1	1	0	1	X	X	X	1	X	X	X	1	X	X	X	X	X
3	0	X	1	0	X	X	X	1	X	X	X	X	X	X	X	X	X	X	X	1	X
4	0	X	X	0	1	1	1	X	X	X	X	X	X	0	X	0	1	0	0	1	X
5	0	X	X	0	X	X	X	1	X	X	X	X	X	X	X	X	X	X	X	X	X
6	X	X	X	0	1	1	1	X	X	1	X	X	X	X	1	1	X	X	1	1	X
7	X	X	X	X	X	X	X	1	1	0	X	X	X	X	X	X	X	X	X	X	X
8	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	0	rd

Table B.4: Control bits for proposed microarchitecture for single precision operation

cycle	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	S0	S1	S2	S3	S4	S5	S6	S7	S8	S9
0	1	1	X	X	X	X	X	X	X	X	X	1	X	X	X	X	X	X	X	X	X
1	1	X	X	1	1	1	1	X	X	X	X	0	0	X	X	1	0	0	0	X	X
2	1	X	X	0	1	1	0	1	X	X	X	1	X	X	X	1	X	X	X	X	X
3	0	X	X	0	1	1	1	1	X	X	X	X	X	X	X	0	X	1	0	1	X
4	0	X	X	0	1	1	0	1	X	X	X	X	X	X	X	0	X	X	X	1	X
5	0	X	1	0	X	X	X	1	X	X	1	X	X	X	X	X	X	X	X	1	X
6	0	X	1	0	1	1	1	X	1	1	0	X	X	1	X	0	1	0	0	1	X
7	X	X	1	0	1	1	1	R11	X	0	X	X	1	X	X	1	0	0	0	0	X
8	X	X	X	X	0	0	1	1	1	1	X	X	X	X	0	X	X	X	1	1	X
9	X	X	X	X	X	X	X	1	1	1	X	X	X	X	X	X	X	X	X	0	X
10	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	0	rd

Table B.5: Control bits for proposed microarchitecture for double precision operation

cycle	RTL	comment
0	$R2 \leftarrow approx(1/B)$	
	$R1 \leftarrow B$	
1	$R7 \leftarrow recode(R2)$	
	$R6 \leftarrow 3 \cdot R1$	$= 3B$
	$R5 \leftarrow R1$	$= B$
	$R4, R1 \leftarrow 2A$ if $A < B$, otherwise A	$= A'$
2	$R6 \leftarrow 3 \cdot R1$	$= 3 \cdot A'$
	$R5 \leftarrow R1$	$= A'$
	$R1 \leftarrow B$	
	$R8 \leftarrow R5 \cdot R7 + \text{INJ}_{RU}(60, 89)$	$= D'_0$
3	$R3 \leftarrow compress(R8)$	$= D'_0$
	$R8 \leftarrow R5 \cdot R7$	$= N'_0$
4	$R5 \leftarrow compress(R8)$	$= N'_0$
	$R6 \leftarrow 3 \cdot R8$	$= 3N'_0$
	$R7 \leftarrow recode(two's\ complement(R3))$	$= F'_0$
5	$R8 \leftarrow R5 \cdot R7 + \text{INJ}_{rnd}(mode)$	$= N'_1$
6	$R7 \leftarrow recode(shift_s(dewpoint(R8)))$	*dewpoint*
	$R5 \leftarrow R1$	$= B$
	$R6 \leftarrow 3 \cdot R1$	$= 3B$
	$R10 \leftarrow compress(R8)$	$= RC_1$
7	$R8 \leftarrow R5 \cdot R7 + (two's\ complement(R4)[22:23])$	$= \alpha$
8	$RC_2 = R10 + 2^{-23}$	compressed RC_2 is ready
	$test(R8[22:47])$	$test(\alpha)$

Table B.6: Register transfer language describing execution of a single precision division in micro-architecture described in Figure B.2

cycle	RTL	comment
0	$R2 \leftarrow approx(1/B)$	
	$R1 \leftarrow B$	
1	$R7 \leftarrow recode(R2)$	
	$R6 \leftarrow 3 \cdot R1$	$= 3B$
	$R5 \leftarrow R1$	$= B$
	$R4, R1 \leftarrow 2A$ if $A < B$, otherwise A	$= A'$
2	$R6 \leftarrow 3 \cdot R1$	$= 3 \cdot A'$
	$R5 \leftarrow R1$	$= A'$
	$R1 \leftarrow B$	
	$R8 \leftarrow R5 \cdot R7 + \text{INJ}_{RU}(60, 89)$	$= D'_0$
3	$R7 \leftarrow recode(ones\text{-}complement(R8))$	$= F'_0$
	$R6 \leftarrow 3 \cdot R8$	$= 3 \cdot D'_0$
	$R5 \leftarrow compress(R8)$	$= D'_0$
	$R8 \leftarrow R5 \cdot R7$	$= N'_0$
4	$R5 \leftarrow compress(R8)$	$= N'_0$
	$R6 \leftarrow 3 \cdot R8$	$= 3N'_0$
	$R8 \leftarrow R5 \cdot R7 + \text{INJ}_{RU}(60, 89)$	$= D'_1$
5	$R8 \leftarrow R5 \cdot R7$	$= N'_1$
	$R3 \leftarrow compress(R8)$	$= D'_1$
	$R11 \leftarrow (R8 < 1)$	for saturated rounding in cycle 7
6	$R7 \leftarrow recode(shift(ones\text{-}complement(R3)))$	$F'_1[27 : 56]$
	$R6 \leftarrow 3 \cdot R8$	$= 3 \cdot N'_1$
	$R3, R5, R10 \leftarrow compress(R8)$	$= N'_1$
7	$R7 \leftarrow recode(R3)$	$= dewpoint^h$
	$R6 \leftarrow 3 \cdot R1$	$= 3 \cdot B$
	$R5 \leftarrow R1$	$= B$
	$R3 \leftarrow 2 - R10$	prepare $(2 - dewpoint^h)$
	if $R11$ then $R8 \leftarrow R5 \cdot R7 + shift(R5) + \text{INJ}_{rnd}(mode)$	$= N''_2$
8	$R10 \leftarrow compress(R8)$	$RC_1 \leftarrow compress(N''_2)$
	$R7 \leftarrow dewpoint\text{-}recode(R8, R3, C_{dew})$	compute & recode $dewpoint^\ell$
	$R8 \leftarrow R5 \cdot R7 + (2 - R4)$	$= \alpha^H, (2 - R4) = 2 - A'$
9	$R8 \leftarrow R5 \cdot R7 + R8[51 : 81]$	$= \alpha^L$
10	$RC_2 = R10 + 2^{-52}$	compressed RC_2 is ready
	$test(R8[51 : 105])$	$test(\alpha^L)$

Table B.7: Register transfer language describing execution of a double precision division in micro-architecture described in Figure B.2

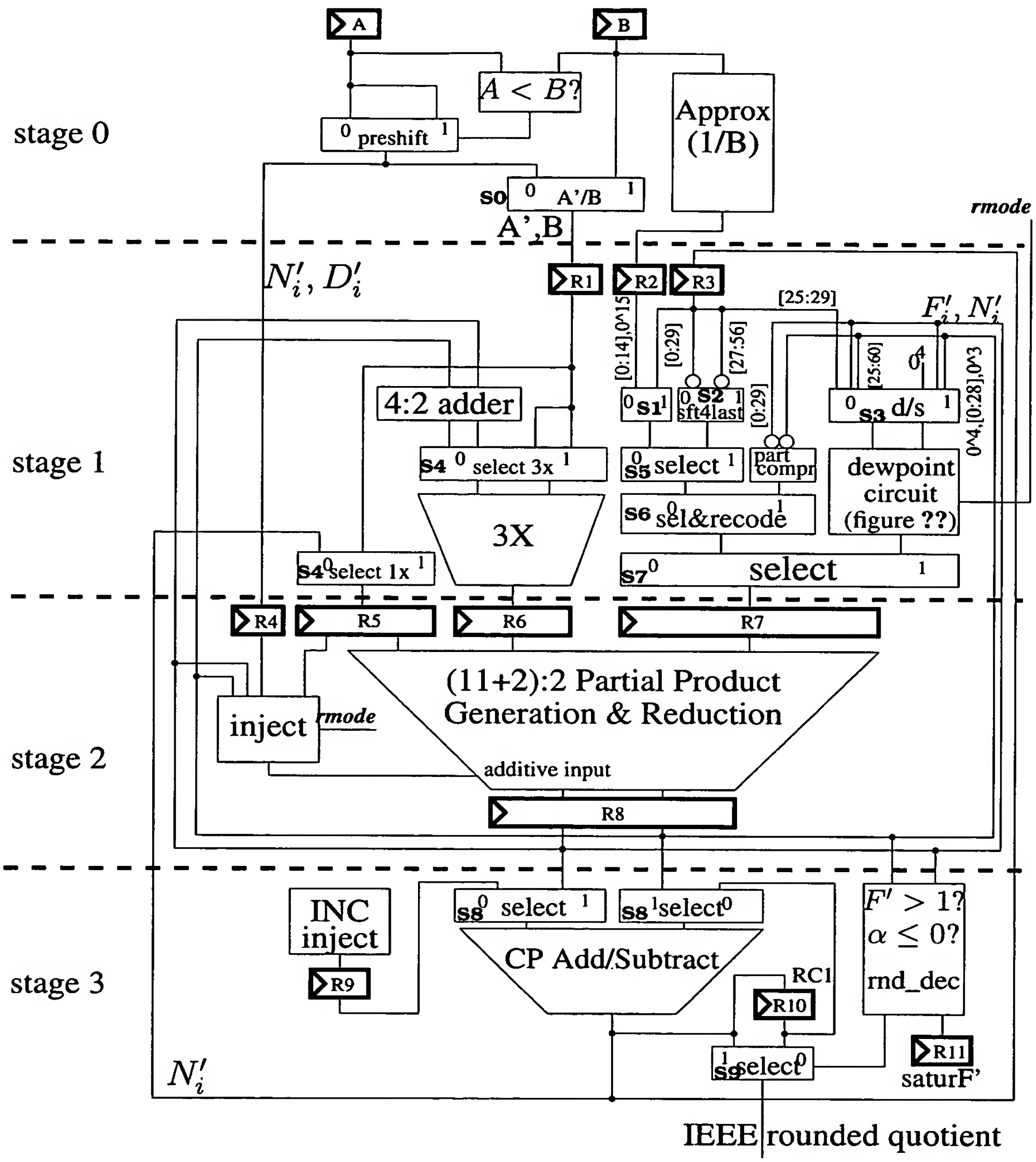

Figure B.2: Microarchitecture for dual mode single precision and double precision Division Implementation.

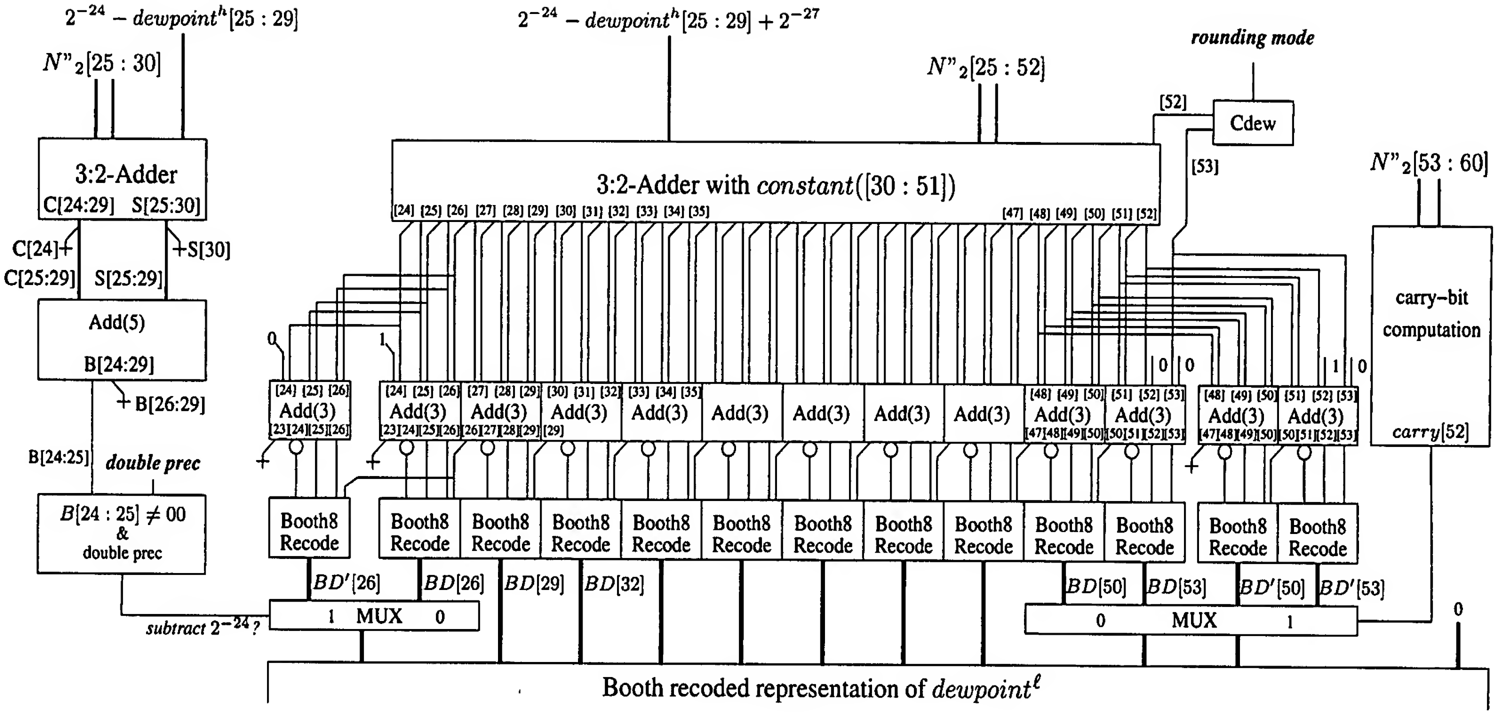

Figure B.3: Implementatqion of dewpoint circuit in pipeline stage 1.

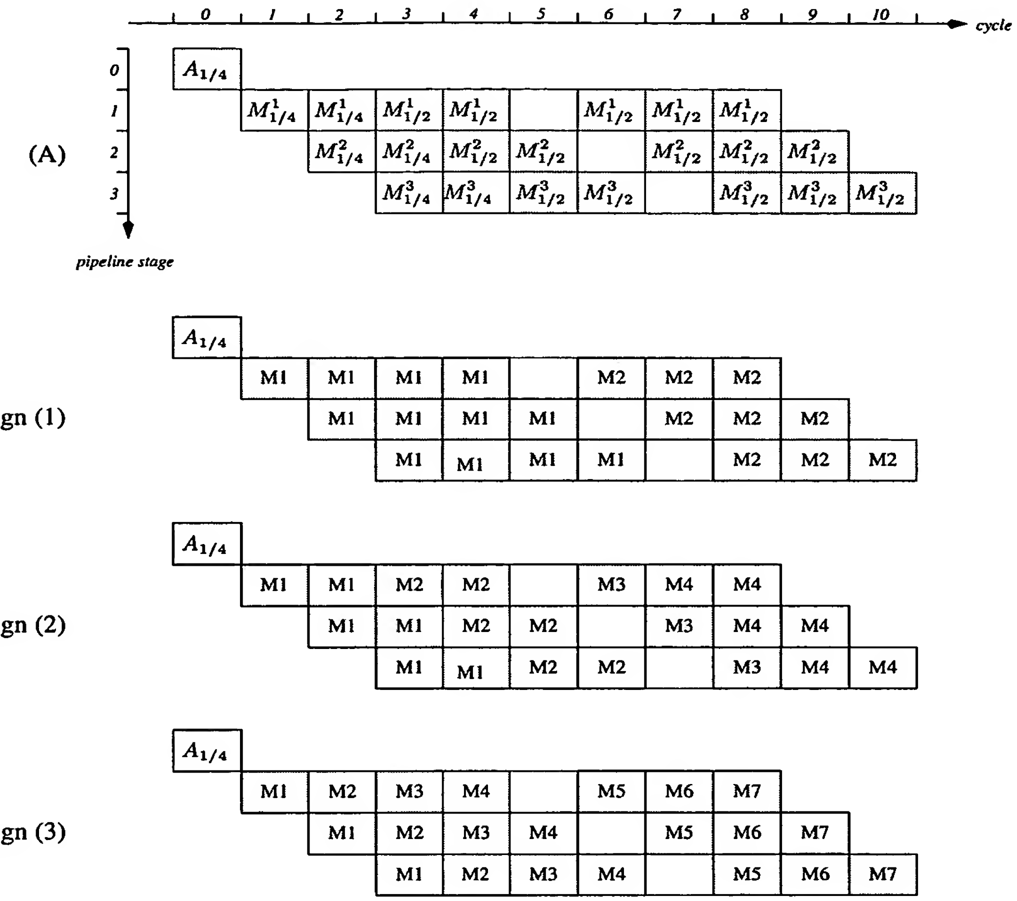

Figure B.4: **(A)** A time-space diagram of double precision division using the micro-architecture depicted in Fig. B.2. $A_{1/4}$ denotes reciprocal approximation with precision roughly $p/4$, $M^i_{1/2}$ denotes the ith pipeline stage of a half-sized multiplier, and $M^i_{1/4}$ denotes the ith pipeline stage of a quarter-sized multiplier. **Proposed design (1)**: M1 and M2 denote two half-sized multipliers. **Proposed design (2)**: M1 denotes a quarter-sized multiplier, and M2-M4 denote half-sized multipliers. **Proposed design (3)**: M1-M2 denote quarter-sized multipliers, and M3-M7 denote half-sized multipliers.

Appendix C

A Parametric Error Analysis of Goldschmidt's Division Algorithm

(Joint work with Warren E. Ferguson. An extended abstract appeared in the proceedings of the 16th IEEE Symposium on Computer Arithmetic. Full version submitted to Journal of Computer and System Sciences.)

abstract

Back in the 60's Goldschmidt presented a variation of Newton-Raphson iterations for division that is well suited for pipelining. The problem in using Goldschmidt's division algorithm is to present an error analysis that enables one to save hardware by using just the right amount of precision for intermediate calculations while still providing correct rounding. Previous implementations relied on combining formal proof methods (that span thousands of lines) with millions of test vectors. These techniques yield correct designs but the analysis is hard to follow and is not quite tight.

We present a simple parametric error analysis of Goldschmidt's division algorithm. This analysis sheds more light on the effect of the different parameters on the error. In addition, we derive closed error formulae that allow to determine optimal parameter choices in four practical settings.

We apply our analysis to show that a few bits of precision can be saved in the floating-point division (FP-DIV) micro-architecture of the AMD-K7$^{\text{TM}}$ microprocessor. These reductions in precision apply to the initial approximation and to the lengths of the multiplicands in the multiplier. When translated to cost, the reductions reflect a savings of 10.6% in the overall cost of the FP-DIV micro-architecture.

C.1 Introduction and Summary

Asymptotically optimal division algorithms are based on multiplicative division methods [23, 30, 37]. Current commercial processor designs employ a parallel multiplier for performing division and square-root operations for floating-point [1, 5, 22, 28]. The parallel multiplier is used for additional operations, such as: multiplication and fused multiply-add. Since meeting the precision requirements of division operations requires more precision than other operations, the dimensions of the parallel multiplier are often determined by precision requirements of division operations. It follows that tighter analysis of the required multiplier dimensions for division operations can lead to improvements in cost, delay, and even power consumption.

The main two methods used for multiplicative division are a variation of Newton's method [14, 15] and a method introduced by Goldschmidt [16] that is based on an approximation of a series expansion. Division based on Newton's method has a quadratic convergence rate (i.e., the number of accurate bits doubles in each iteration) and is self-correcting (i.e., inaccuracies of intermediate computations do not accumulate). A rigorous error analysis of Newton's method appears in [3, 20, 25] and for various exceptional cases in [5]. The analysis in [3, 20] considers the smallest precision required per iteration. Our error analysis follows this spirit by defining separate error parameters for every intermediate computation. In addition, the analysis in [3, 20] relies on directed roundings, a method that we use as well.

Each iteration of Newton's method involves two *dependent* multiplications; namely, the product of the first multiplication is one of the operands of the second multiplication. The implication of having to compute two dependent multiplications per iteration is that these multiplications cannot be parallelized or pipelined.

Goldschmidt's division algorithm also requires two multiplication per iteration and the convergence rate is the same as Newton's method. However, the most important feature of Goldschmidt's algorithm is that the two multiplications per iteration are *independent* and can be pipelined or computed in parallel. On the other hand, Goldschmidt's algorithm is not self-correcting; namely, inaccuracies of intermediate computations accumulate and cause the computed result to drift away from the accurate quotient. Goldschmidt's division algorithm was used in the IBM System/360 model 91 [2] and even more recently in the IBM S/390 [33] and in the AMD-K7$^{\text{TM}}$ microprocessor [28]. However, lack of a general and simple error analysis of Goldschmidt's division algorithm has averted most designers from considering implementing Goldschmidt's algorithm. Thus most implementations of multiplicative division methods have been based on Newton's method in spite of the longer latency due to dependent multiplications in each iteration [5, 22] (see also [38] for more references).

Goldschmidt's method is not self-correcting as explained in [21] (there is a wrong comment on this in [39]). This makes it particularly important and difficult to keep track of accumulated and propagated error terms during intermediate computations. We were not able to locate a general analysis of error bounds of Goldschmidt's algorithm in the literature. Goldschmidt's error analysis in [16] is with respect to a design that uses a serial radix-4 Booth multiplier with 61-bits. Goldschmidt's design computes the quotient of two binary numbers in the range $[1/2, 1)$, and his analysis shows that the absolute error is in the range $[-2^{56}, 0]$. Krishnamurthy [21] analyzes the error only for the case that only one multiplicand is allowed to be imprecise in intermediate computations (the second multiplicand must be precise); such an analysis is only useful for determining lower bounds for delay. Recent implementations of Goldschmidt's division algorithm still rely on an error analysis that over-estimates the accumulated error [28]. Such over-estimates lead to correct designs but waste hardware and cause unnecessary delay (since the multiplier and the initial lookup table are too large). These over-estimations were based on informal arguments that were confirmed by a mechanically verified proof that spans over 250 definitions and 3000 lemmas [31].

Agarwal *et al.* [1] presented a multiplicative division algorithm that is based on an approximate series expansion. This algorithm was implemented in IBM's Power3™. Their algorithm provides no advantages over Goldschmidt's algorithm. In double precision, their algorithm requires 8 multiplications and the longest chain of dependent multiplications consists of 4 multiplications.

We present a version of Goldschmidt's division algorithm that uses directed roundings. We develop a simple general parametric analysis of tight error bounds for our version of Goldschmidt's division algorithm. Our analysis is parametric in the sense that it allows arbitrary one-sided errors in each intermediate computation and it allows an arbitrary number of iterations. In addition, we suggest four practical simplified settings in which errors in intermediate computations are not arbitrary. For each of these four settings, we present a closed error formula. The advantage of closed formulae is in simplifying the task of finding optimal parameter combinations in implementations of Goldschmidt's division method for a given target precision.

We demonstrate the advantages of our error analysis by showing how it could lead to savings in cost and delay. For this purpose we consider Oberman's [28] floating-point micro-architecture used in the AMD-K7™ design. We present a micro-architecture that implements our version of Goldschmidt's algorithm and follows the micro-architecture described in [28]. The modules building our micro-architecture were made as similar as possible to the modules in [28]. This was done so that the issue of the precisions of the lookup table and multiplier could be isolated from other issues. Based on our analysis, we use a smaller multiplier (70×74 bits compared to 76×76 in [28]) and we allow a slightly larger initial error ($2^{-13.51}$ compared to $2^{-13.75}$ in [28]). Based on the cost models of Paul & Seidel [29] and Mueller & Paul [25], we estimate that our parameter choices for multiplier widths and initial approximation accuracy reduce the cost of the micro-architecture by 10.6% compared to the parameter choices in [28].

The paper is organized as follows. In Section C.2 we present Newton's method for division and then proceed by presenting Goldschmidt's algorithm as a variation of Newton's method. In Section C.3, a version of Goldschmidt's algorithm with imprecise intermediate computations is presented as well as an error analysis. In Section C.4 we develop closed form error bounds for Goldschmidt's method with respect to four specific settings. In Section C.5 we present an alternative micro-architecture to [28] and compare costs.

C.2 Goldschmidt's division algorithm

In this section we present Newton's method for computing the reciprocal of a given number. We then continue by describing a version of Goldschmidt's division algorithm [16] that uses precise intermediate computations. We show how Goldschmidt's algorithm is derived from Newton's method. The error analysis of Newton's method is used to analyze the errors in Goldschmidt's algorithm.

C.2.1 Newton's method.

Newton's method can be applied to compute the reciprocal of a given number. To compute the reciprocal of $B > 0$, apply Newton's method to the function $f(x) = B - 1/x$. Note that: (a) the root of $f(x)$ is $1/B$, which is the reciprocal we want to compute, and (b) the function $f(x)$ has a derivative $f'(x) = x^{-2}$ in the interval $(0, \infty)$. In particular, the derivative $f'(x)$ is positive.

Newton iterations are defined by the following recurrence: Let x_0 denote an initial estimate $x_0 \neq 0$ and define x_{i+1} by

$$\begin{aligned} x_{i+1} &= x_i - \frac{f(x_i)}{f'(x_i)} \\ &= x_i - \frac{B - \frac{1}{x_i}}{x_i^{-2}} \\ &= x_i \cdot (2 - B \cdot x_i). \end{aligned} \tag{C.1}$$

A few iteration steps are visualized in Figure C.1.

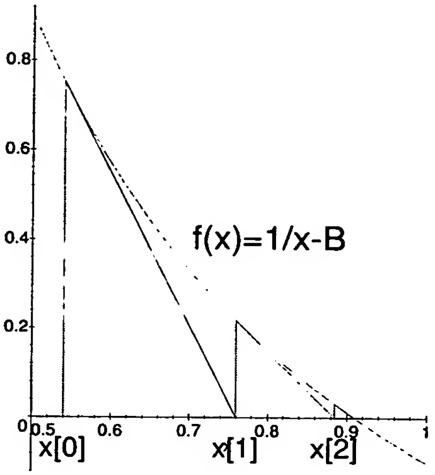

Figure C.1: The first iteration steps of the Newton approximation of $1/B$.

Consider the *relative error* term e_i defined by

$$\begin{aligned} e_i &\triangleq \frac{\frac{1}{B} - x_i}{\frac{1}{B}} \\ &= 1 - B \cdot x_i. \end{aligned}$$

It follows that

$$\begin{aligned} e_{i+1} &= 1 - B \cdot x_{i+1} \\ &= 1 - B \cdot x_i \cdot (2 - B \cdot x_i) \\ &= (1 - B \cdot x_i)^2 \\ &= e_i^2. \end{aligned} \tag{C.2}$$

Equation C.2 has three implications:

1. Convergence of x_i to $1/B$ at a quadratic rate is guaranteed provided that the initial relative error is less than 1. Equivalently, convergence holds if $x_0 \in (0, \frac{2}{B})$.

2. For $i \geq 1$, the relative error e_i is non-negative, hence, $x_i \leq 1/B$. This property is referred to

Algorithm 3 Goldschmidt-Divide(A, B) - Goldschmidt's iterative algorithm for computing A/B

```
Require: $|e_0| < 1$.
Initialize: $N_{-1} \leftarrow A$, $D_{-1} \leftarrow B$, $F_{-1} \leftarrow \frac{1-e_0}{B}$.
for $i = 0$ to $k$ do
    $N_i \leftarrow N_{i-1} \cdot F_{i-1}$.
    $D_i \leftarrow D_{i-1} \cdot F_{i-1}$.
    $F_i \leftarrow 2 - D_i$.
end for
Return($N_i$)
```

as *one-sided convergence*.

3. If $B \in [1, 2)$, then also the ***absolute error*** decreases at a quadratic rate. Hence, the number of "accurate" bits doubles in every iteration, and the number of iterations required to obtain p bits of accuracy is logarithmic in p.

The disadvantage of Newton's iterations, with respect to a pipelined multiplier, is that each iteration consists of 2 *dependent* multiplications: $\alpha_i = B \cdot x_i$ and $\beta_i = x_i \cdot (2 - \alpha_i)$. Namely, the product β_i cannot be computed before the product α_i is computed.

C.2.2 Goldschmidt's Algorithm

In this section we describe how Goldschmidt's algorithm can be derived from Newton's method. Here, our goal is to compute the quotient A/B. Goldschmidt's algorithm uses three values N_i, D_i and F_i defined as follows:

$$N_i \triangleq A \cdot x_i$$

$$D_i \triangleq B \cdot x_i$$

$$F_i \triangleq 2 - D_i.$$

Consider Newton's iteration: $x_{i+1} = x_i \cdot (2 - B \cdot x_i)$. We may rewrite an iteration by

$$x_{i+1} = x_i \cdot F_i,$$

which when multiplied by A and B, respectively, becomes

$$N_{i+1} = N_i \cdot F_i \overset{i \to \infty}{\longrightarrow} A/B$$
$$D_{i+1} = D_i \cdot F_i \overset{i \to \infty}{\longrightarrow} 1.$$

Since x_i converges to $1/B$, it follows that N_i converges to A/B and D_i converges to 1.

Note that: (a) $N_i/D_i = A/B$, for every i; (b) N_i converges to A/B at the same rate that x_i converges to $1/B$; and (c) Let $A, B > 0$. Since the relative error e_i in Newton's is non-negative, for $i \geq 1$, it follows that $N_i \leq A/B$, $D_i \leq 1$ and $F_i \geq 1$ for $i \geq 1$.

As in Newton's iterations, the algorithm converges if $|e_0| < 1$ and the relative error decreases quadratically. One could use a fixed initial approximation of the quotient. Usually a more accurate initial approximation of $1/B$ is computed by a lookup table or even a more elaborate functional unit (c.f. [6, 36]).

Algorithm 3 lists Goldschmidt's division algorithm. Given A and B the algorithm computes the quotient A/B. The listing uses the same notation used above, and iterates k times.

Observe that the two multiplications that take place in every iteration (in Lines 3-4) are independent, and therefore, Goldschmidt's division algorithm is more amenable to pipelined implementations. Figure C.2 depicts and compares the flowcharts of the iterations of Newton's method and Goldschmidt's algorithm. The initial approximation is assumed to depend on the value of B. A closer look at Figure C.2 reveals that k iterations of either Newton's method or Goldschmidt's algorithm require $2k + 1$ multiplications (the unfilled boxes in the figure refer to initializations in which multiplication does not take place). These $2k + 1$ multiplication must be linearly ordered in Newton's method implying a critical path of $2k + 1$ dependent multiplications. In Goldschmidt's algorithm the two multiplications that take place in every iterations are independent, hence the critical path consists only of $k + 1$ multiplications.

An error analysis of Goldschmidt's algorithm with precise arithmetic is based on the following claim.

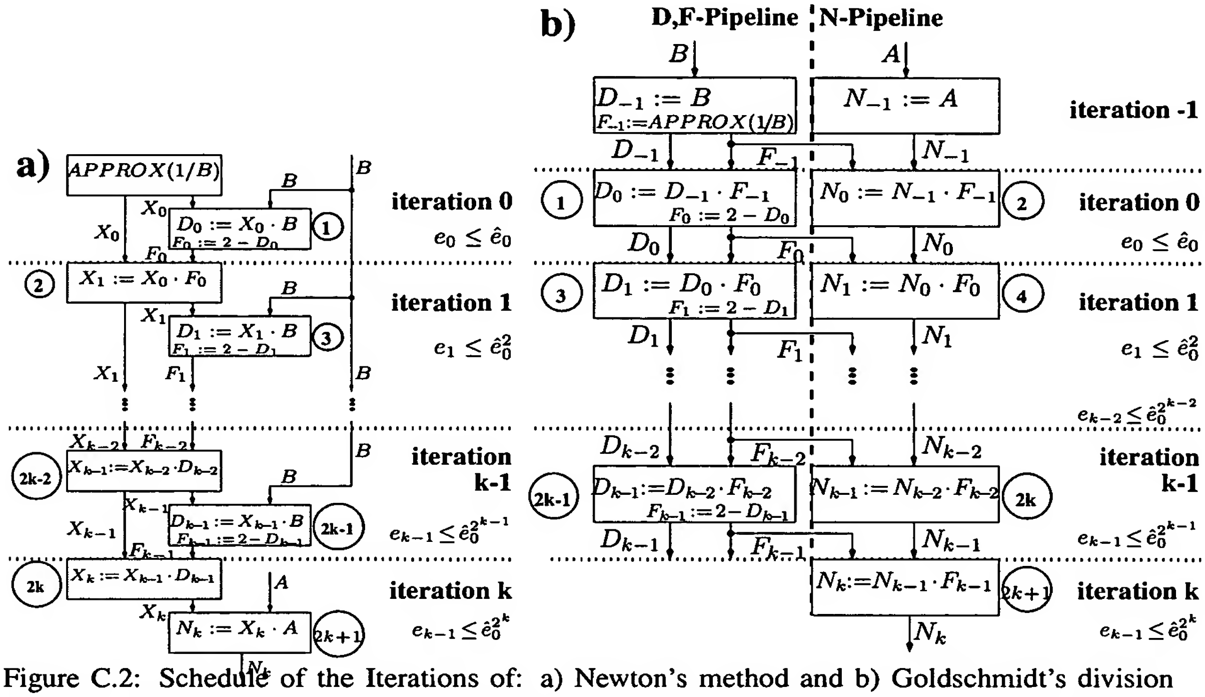

Figure C.2: Schedule of the Iterations of: a) Newton's method and b) Goldschmidt's division algorithm using an initial approximation for $1/B$. The numbers in circles indicate the sequence of the multiplications involved. A bound on the relative error e_i of iteration i appears in each iteration.

Claim 28 *The following equalities hold for $i \geq 0$:*

$$D_i = 1 - e_0^{2^i} \tag{C.3}$$

$$F_i = 1 + e_0^{2^i} \tag{C.4}$$

The key difficulty in analyzing the error in imprecise implementations of Goldschmidt's algorithm is due to the violation of the invariant $N_i/D_i = A/B$. Consider the equality

$$\begin{aligned} N_k &= A/B \cdot D_0 \cdot F_0 \cdot F_1 \cdot \ \ldots\ \cdot F_{k-1} \\ &= A/B \cdot (1 - e_0) \cdot (1 + e_0) \cdot (1 + e_0^2) \cdot (1 + e_0^4) \cdot \ \ldots\ \cdot (1 + e_0^{2^{k-1}}). \end{aligned}$$

Imprecise $D_0, F_0, \ldots, F_{k-1}$ accumulate to an imprecise approximation of A/B.

How many iterations are required? Consider an initial relative error of $|e_0| \leq 2^{-a}$, and assume that a quotient (reciprocal) with a relative error smaller than 2^{-n} is required. Implementations with precise intermediate computations of Newton's method and Goldschmidt's algorithm require $k = \lceil log_2(n/a) \rceil$ iterations.

Precision of intermediate multiplications. Let $len(x)$ denote the number of bits in the binary representation of x. Precisions above $2n$ (where n denotes the number of bits used to represent each of the operands A and B) are usually considered too high for practical implementations. The following claim deals with the lengths of the operands in intermediate calculations of Goldschmidt's algorithm. The claim shows that the required precision is $(a+n) \cdot \frac{n}{a}$. This renders implementations with precise arithmetic impractical.

Claim 29 *Let* $\text{len}(A) = \text{len}(B) = n$ *and* $\text{len}(F_{-1}) = a$. *The lengths of the operands* N_i, D_i *and* F_i *in Goldschmidt's algorithm are* $2^i \cdot (a + n)$, *for* $i \geq 0$.

C.3 Imprecise Intermediate Computations

This section contains the core contribution of our paper. We present a version of Goldschmidt's algorithm with imprecise intermediate computations. In this algorithm the invariant $N_i/D_i = A/B$ of Goldschmidt's algorithm with precise arithmetic does not hold anymore. We then develop a simple parametric analysis for error bounds in this algorithm. The error analysis is based on relative errors of intermediate computations. The setting is quite general and allows for different relative errors for each computation.

We define the relative error as follows.

Definition 7 *The* relative error *of x with respect to y is defined by*

$$\frac{y - x}{y}.$$

Note that one usually uses the negative definition (i.e., $(x - y)/y$). We prefer this definition since is helps clarify the direction of the directed roundings that we use.

The analysis begins by using the exact values of all the relative errors. The values of the relative errors depend on the actual values of the inputs and on the hardware used for the intermediate computations. However, one can usually easily derive upper bounds on the absolute errors of each intermediate computation. E.g., such bounds on the absolute errors are simply derived from the precision of the multipliers. Our analysis continues by translating the upper bounds on the absolute errors to upper bounds on the relative errors. Hence we are able to analyze the accuracy of an implementation of the proposed algorithm based on upper bounds on the absolute errors.

An interesting feature of our analysis is that directed roundings are used for all intermediate calculations. Surprisingly, directed roundings play a crucial role in this analysis and enable a simpler and tighter error analysis than round-to-nearest rounding (c.f. [28]).

C.3.1 Goldschmidt's division algorithm using approximate arithmetic

A listing of Goldschmidt's division algorithm using approximate arithmetic appears in Algorithm 4. The values corresponding to N_i, D_i, and F_i using the imprecise computations are denoted by N'_i, D'_i and F'_i, respectively.

Directed roundings are used for all intermediate calculations. For example, N'_i is obtained by rounding down the product $N'_{i-1} \cdot F'_{i-1}$. We denote by n_i the relative error of N'_i with respect to $N'_{i-1} \cdot F'_{i-1}$. Since $N'_{i-1} \cdot F'_{i-1}$ is rounded down, we assume that $n_i \geq 0$. Similarly, rounding down is used for computing F'_i (with the relative error f_i) and rounding up is used for computing D'_i (with the relative error d_i).

The initial approximation of the reciprocal $1/B$ is denoted by F'_{-1}. The relative error of F'_{-1} with respect to $1/B$ is denoted by e_0. We do not make any assumption about the sign of e_0.

Our error analysis is based on the following assumptions:

1. The operands are in the range $A, B \in [1, 2)$.

2. All the relative errors incurred by directed rounding are at most $1/4$. This assumption is easily met by multipliers with more than 4 bits of precision.

3. We require that $|e_0| + 3d_0/2 + f_0 < 1/2$. Again, this assumption is easily met if the multiplications and the initial reciprocal approximation are precise enough.

4. The initial approximation F'_{-1} of $1/B$ is in the range $[1/2, 1]$. This assumption is easily met if lookup tables are used.

Algorithm 4 Goldschmidt-Approx-Divide(A, B) - Goldschmidt's division algorithm using approximate arithmetic

```
Initialize: $N'_{-1} \leftarrow A$, $D'_{-1} \leftarrow B$, $F'_{-1} \leftarrow \frac{1-e_0}{B}$.
for $i = 0$ to $k$ do
    $N'_i \leftarrow (1 - n_i) \cdot N'_{i-1} \cdot F'_{i-1}$.
    $D'_i \leftarrow (1 + d_i) \cdot D'_{i-1} \cdot F'_{i-1}$.
    $F'_i \leftarrow (1 - f_i) \cdot (2 - D'_i)$.
end for
Return$(N'_i)$
```

Definition 8 *The* relative error *of N'_i with respect to A/B is*

$$\rho(N'_i) \triangleq \frac{A/B - N'_i}{A/B}.$$

C.3.2 A simplifying assumption: strict directed roundings

The following assumption about directed rounding used in Algorithm 4 helps simplify the analysis.

Definition 9 (Strict Directed (SD) rounding) *Rounding down is* strict *if $x \geq 1$ implies that $rnd(x) \geq 1$. Similarly, rounding up is* strict *if $x \leq 1$ implies that $rnd(x) \leq 1$.*

Observe that, in general, rounding down means that $rnd(x) \leq x$, for all x. Often the absolute error introduced by rounding is bounded by $\varepsilon > 0$, namely $x - \varepsilon \leq rnd(x) \leq x$. Strict rounding down requires that if $x \geq 1$, then $rnd(x) \geq 1$ no matter how close x is to 1. In non-redundant binary representation strict rounding is easily implemented as follows. Strict rounding down can be implemented by truncating. Strict rounding up can be obtained by (i) an increment by a unit in each position below the rounding position and (ii) truncation of the bit string in positions less significant than the rounding position.

Assumption 30 (SD rounding) *All directed roundings used in Algorithm 4 are strict.*

C.3.3 Parametric Error Analysis

Lemma 31 bounds the ranges of N'_i and F'_i in Algorithm 4 under Assumption 30. This lemma is an extension of the properties $D_i \leq 1$ and $F_i \geq 1$ (for $i \geq 1$) of Algorithm 3.

Goldschmidt already pointed out that since F_i tends to 1 from above, one could save hardware since the binary representation of F_i begins with the string 1.000.... An analogous remark holds for D_i. However, Lemma 31 refers to the inaccurate intermediate results (i.e., D'_i and F'_i) rather than the precise intermediate results (i.e., D_i and F_i). Parts 2-3 of lemma 31 show that the same hardware reduction strategy applies to Algorithm 4, even though intermediate calculations are imprecise.

Definition 10 *Define δ_i, for $i \geq 0$, as follows:*

$$\delta_i \quad := \quad \begin{cases} |e_0| + 3d_0/2 & \textit{for } i = 0 \\ \delta_{i-1}^2 + f_{i-1} & \textit{otherwise.} \end{cases}$$

Lemma 31 *(ranges of D'_i and F'_i)*

The following bounds hold:

1. $D'_0 \in [1 - \delta_0, 1 + \delta_0] \subseteq (1/2, 3/2)$.

2. $D'_i \in [1 - \delta_i, 1]$, *for every $i \geq 1$.*

3. $F'_i \in [1, 1 + \delta_i]$, *for every $i \geq 1$.*

4. $D'_i \leq D'_{i+1}$, *for every* $i \geq 1$.

Proof: Part (1): Since $D'_0 = (1 + d_0) \cdot D'_{-1} \cdot F'_{-1}$, it follows that $D'_0 = (1 + d_0) \cdot (1 - e_0) = 1 - e_0 + d_0(1 - e_0)$. Since $d_0 > 0$ and $|e_0| < 1/2$, it follows that $(1 - e_0) < 3/2$ and

$$1 - \delta_0 \;\leq\; 1 - |e_0| \;\leq\; D'_0 \;\leq\; 1 + (|e_0| + 3d_0/2) \;\leq\; 1 + \delta_0.$$

The bounds of $1/2 < 1-\delta_0 \leq 1+\delta_0 < 3/2$ follow immediately from the condition $0 \leq \delta_0 \leq |e_0| + 3d_0/2 + f_0 < 1/2$.

Part (2): We prove that if $D_{i-1} \in [1 - \delta_{i-1}, 1 + \delta_{i-1}]$, then $D_i \in [1 - \delta_i, 1]$. It then follows from Part (1) that D_1 is in the required range, and then by induction, D_i is in the required range, for every $i \geq 1$.

Let $t_{i-1} = D_{i-1} - 1$. The assumption $D_{i-1} \in [1 - \delta_{i-1}, 1 + \delta_{i-1}]$ implies that $|t_{i-1}| \leq \delta_{i-1}$. The value of D'_i can then be written as:

$$\begin{aligned}
D'_i &= (1 + d_i) \cdot \underbrace{(1 + t_{i-1})}_{D'_{i-1}} \cdot \underbrace{(1 - t_{i-1}) \cdot (1 - f_{i-1})}_{F'_{i-1}} \\
&= \underbrace{(1 + d_i)}_{\geq 1} \cdot \underbrace{(1 - t^2_{i-1})}_{\leq 1} \cdot \underbrace{(1 - f_{i-1})}_{\leq 1} \\
&\geq (1 - t^2_{i-1}) \cdot (1 - f_{i-1}) \qquad\qquad \text{(C.5)} \\
&\geq (1 - t^2_{i-1} - f_{i-1}) \\
&\geq (1 - \delta^2_{i-1} - f_{i-1}) \\
&= 1 - \delta_i
\end{aligned}$$

The multiplication by $(1 + d_i)$ in the second line of Equation C.5 models the strict rounding up of $(1 - t_i^2) \cdot (1 - f_i) \leq 1$. Hence, $D'_i \leq 1$, and Part (2) follows.

Part (3): By Part (2), $1 - \delta_i \leq D'_i \leq 1$, hence $1 \leq 2 - D'_i \leq 1 + \delta_i$. Recall that F'_i is obtained by strict rounding down of $2 - D'_i$, hence $1 \leq F'_i \leq 1 + \delta_i$.

Part (4): From Part (3) it follows that $F'_i \geq 1$. Since D'_{i+1} is obtained by rounding up $D'_i \cdot F'_i$, it follows that $D'_{i+1} \geq D'_i$, as required. □

The following claim summarizes the relative error of Goldschmidt's division algorithm using approximate arithmetic.

Theorem 32 *For every $i > 0$, the relative error $\rho(N'_i) = \frac{A/B - N'_i}{A/B}$ satisfies*

$$\pi_i \;\leq\; \rho(N'_i) \;\leq\; \pi_i + \delta_i, \tag{C.6}$$

where π_i is defined by $\pi_i \triangleq 1 - (1 - n_i) \cdot \prod_{j=0}^{i-1} \frac{1-n_j}{1+d_j} \;\geq 0$.

Proof: We decompose $\rho(N'_i)$ as follows.

$$\rho(N'_i) = \frac{A/B - N'_i}{A/B} = \frac{A/B - N'_i/(D'_{i-1} \cdot F'_{i-1})}{A/B} + \frac{N'_i/(D'_{i-1} \cdot F'_{i-1}) - N'_i}{A/B}. \tag{C.7}$$

We first show that the first term on the right hand side equals π_i. Observe that $\frac{N'_i}{D'_i}$ can be expanded as follows:

$$\begin{aligned} \frac{N'_i}{D'_i} &= \frac{N'_{i-1}}{D'_{i-1}} \cdot \frac{1 - n_i}{1 + d_i} \\ &= \frac{A}{B} \cdot \prod_{j=0}^{i} \frac{1 - n_j}{1 + d_j}. \end{aligned} \tag{C.8}$$

Equation C.8 implies that N'_i/D'_i is non-increasing.

An expansion of $\frac{N'_i}{D'_{i-1} \cdot F'_{i-1}}$ yields

$$\begin{aligned} \frac{N'_i}{D'_{i-1} \cdot F'_{i-1}} &= \frac{(1 - n_i)N'_{i-1} \cdot F'_{i-1}}{D'_{i-1} \cdot F'_{i-1}} && \text{by definition of } N'_i \\ &= (1 - n_i) \cdot \frac{N'_{i-1}}{D'_{i-1}} \\ &= (1 - n_i) \cdot \frac{A}{B} \cdot \prod_{j=0}^{i-1} \frac{1 - n_j}{1 + d_j} && \text{by Eq. C.8.} \end{aligned}$$

Note that in particular $\frac{N'_i}{D'_{i-1} \cdot F'_{i-1}} \leq \frac{A}{B}$, hence $\rho(\frac{N'_i}{D'_{i-1} \cdot F'_{i-1}})$ is non-negative. It follows that

$\rho(\frac{N'_i}{D'_{i-1} \cdot F'_{i-1}})$ satisfies

$$\begin{aligned}\rho(\frac{N'_i}{D'_{i-1} \cdot F'_{i-1}}) &= \frac{A/B - N'_i/(D'_{i-1} \cdot F'_{i-1})}{A/B} \\ &= 1 - (1 - n_i) \cdot \prod_{j=0}^{i-1} \frac{1 - n_j}{1 + d_j} \qquad \text{(C.9)} \\ &= \pi_i.\end{aligned}$$

We now prove that the second term on the right hand side of Eq. C.7 is non-negative and bounded by δ_i. Consider the numerator

$$\begin{aligned}\frac{N'_i}{D'_{i-1} \cdot F'_{i-1}} - N'_i &= \frac{N'_i}{D'_{i-1} \cdot F'_{i-1}} \cdot (1 - D'_{i-1} \cdot F'_{i-1}) \\ \text{(by eq. C.9)} \quad &= (1 - \pi_i) \cdot \frac{A}{B} \cdot (1 - D'_{i-1} \cdot F'_{i-1}) \\ &\geq 0.\end{aligned}$$

To complete the proof we only need to show that $0 \leq 1 - D'_{i-1} \cdot F'_{i-1} \leq \delta_i$. By Parts 1-2 of Claim 31, it follows that $D'_{i-1} \in [1 - \delta_{i-1}, 1 + \delta_{i-1}]$ and $F'_{i-1} = (1 - f_{i-1}) \cdot (2 - D'_{i-1})$. Hence $D'_{i-1} \cdot F'_{i-1}$ is bounded by

$$1 - \delta_i \quad \leq \quad (1 - f_{i-1}) \cdot (1 - \delta^2_{i-1}) \quad \leq \quad D'_{i-1} \cdot F'_{i-1} \quad \leq \quad D'_{i-1} \cdot (2 - D'_{i-1}) \quad \leq \quad 1,$$

and the claim follows. □

For $i = 0$ it can be verified that $|\rho(N'_0)| \leq \pi_0 + \delta_0$.

A somewhat looser (yet easier to evaluate) bound on the relative error follows from

$$\pi_i \leq \sum_{j=0}^{i} n_j + \sum_{j=0}^{i-1} d_j. \qquad \text{(C.10)}$$

The proof of equation C.10 is as follows. Note that $(1 + x)^{-1} \geq 1 - x$, for $x \in [0, 1)$ and

$\prod_i(1-x_i) \geq 1 - \sum_i x_i$, for $x_i \in [0,1)$. Hence,

$$\begin{aligned}\pi_i &= 1-(1-n_i)\cdot\prod_{j=0}^{i-1}\frac{1-n_j}{1+d_j}\\ &\leq 1-(1-n_i)\cdot\prod_{j=0}^{i-1}(1-n_j)\cdot(1-d_j)\\ &\leq 1-\left(1-\left(n_i+\sum_{j=0}^{i-1}(n_j+d_j)\right)\right)\\ &= n_i+\sum_{j=0}^{i-1}(n_j+d_j).\end{aligned}$$

C.3.4 Deriving bounds on relative errors from absolute errors

In this subsection we obtain bounds on the relative errors $\rho(N_i')$ from the absolute errors of the intermediate computations. The reason for doing so is that in an implementation one is able to easily bound the absolute errors of intermediate computations; these follow directly from the precision of the operation, the rounding used (e.g., floor or ceiling), and the representation of the results (binary, carry-save, etc.).

Consider the computation of N_i'. The relative error introduced by this computation is n_i, and N_i' equals $(1-n_i)\cdot N_{i-1}'\cdot F_{i-1}'$. An accurate computation would produce the product $N_{i-1}'\cdot F_{i-1}'$. Hence, the absolute error is $n_i\cdot N_{i-1}'\cdot F_{i-1}'$.

Definition 11 *The absolute errors of intermediate computations are defined as follows:*

$$\begin{aligned}neps_i &\triangleq n_i\cdot N_{i-1}'\cdot F_{i-1}'\\ deps_i &\triangleq d_i\cdot D_{i-1}'\cdot F_{i-1}'\\ feps_i &\triangleq f_i\cdot(2-D_i').\end{aligned}$$

In an implementation, the exact absolute errors are unknown. Instead, we use upper bounds on the absolute errors. We denote these upper bounds as follows: $\widehat{neps}_i \geq neps_i$, $\widehat{deps}_i \geq deps_i$ and $\widehat{feps}_i \geq feps_i$.

The following claim shows how one can derive upper bounds on the relative errors from upper bounds on the absolute errors.

Claim 33 *(from absolute to relative errors) If $A, B \in [1,2)$, then for $i \geq 2$ the relative errors are bounded by:*

$$\begin{array}{rclcl} 0 & \leq & n_i & \leq & 2\widehat{neps}_i/(1-\pi_{i-1}-\delta_{i-1}) \\ 0 & \leq & d_i & \leq & \widehat{deps}_i/(1-\delta_{i-1}) \\ 0 & \leq & f_i & \leq & \widehat{feps}_i \end{array}$$

Proof: It follows from Definition 11 that

$$n_i = \frac{neps_i}{N'_{i-1} \cdot F'_{i-1}}, \qquad d_i = \frac{deps_i}{D'_{i-1} \cdot F'_{i-1}}, \qquad f_i = \frac{feps_i}{2-D'_i}.$$

By Theorem 32 it follows that, for $i-1 \geq 1$,

$$N'_{i-1} \geq \frac{A}{B} \cdot (1-\pi_{i-1}-\delta_{i-1}).$$

By Part 3 of Lemma 31 it follows that, for $i-1 \geq 1$, $F'_{i-1} \geq 1$. Hence

$$n_i \leq \widehat{neps}_i \cdot \frac{2}{1-\pi_{i-1}-\delta_{i-1}}.$$

By Parts 2 and 3 of Lemma 31 it follows that $1 \geq D'_{i-1} \geq 1-\delta_{i-1}$ and $F'_{i-1} \geq 1$. Hence the bounds on d_i and f_i follow. □

A careful reader might be concerned by the fact that δ_{i-1} and π_{i-1} appear in the above bounds on the relative errors n_i and d_i. When analyzing the errors, one computes upper bounds for all relative errors from the first iteration to the last. These bounds are used to compute upper bounds on δ_{i-1} and π_{i-1}, which in turn are used to bound n_i and d_i. In section C.6, bounds are given for the relative errors in the first and second iteration (see Claim 37 in Section C.6).

C.4 Closed Form Error Bounds in Specific Settings

In this section we describe four settings of the relative errors that enable us to derive closed form error bounds. The advantage of having closed form error bounds is that such bounds simplify the task of minimizing an objective function (modeling cost or delay) subject to the required precision. Closed form error bounds also enable one to easily evaluate the effect of design choices (e.g., initial error, precision of intermediate computations, and number of iterations) on the final error.

C.4.1 Setting I: $n_i, d_i \leq \hat{n}$ and $f_i = 0$.

Setting I deals with the situation that all the relative errors n_i, d_i are bounded by the same value $\hat{n}$. In addition it is assumed in this setting that $f_i = 0$, for every i. The justification for Setting I is that if all internal operands are represented by binary strings of equal length, then it is possible to bound all the relative errors n_i, d_i by the same value. The relative errors f_i can be assumed to be 0, if the computations $F'_i = (2 - D'_i)$ are precise.

Using Theorem 32 and Equation C.10, the relative approximation error $\rho(N'_i) = \frac{A/B - N'_i}{A/B}$ in Setting I can be bounded by:

$$0 \leq \rho(N'_i) = \frac{A/B - N'_i}{A/B} \leq (2i+1) \cdot \hat{n} + (|e_0| + 3\hat{n}/2)^{2^i}.$$

C.4.2 Setting II: $n_i, d_i \leq \hat{n}$ and f_i/δ_i^2 is exponential in $-k$.

In setting II it is assumed that $f_i/\delta_i^2 \leq \alpha \cdot 2^{-k}$, where α is an appropriately chosen constant and k is the number of iterations. In addition, it is assumed that $n_i, d_i \leq \hat{n}$, for every i.

Claim 34

$$\begin{aligned}\delta_k &\leq \delta_0^{2^k} \cdot (1 + \alpha \cdot 2^{-k})^{2^k - 1} \\ &\leq \delta_0^{2^k} \cdot e^{\alpha}.\end{aligned}$$

Observe that in Setting II δ_k converges quadratically and that δ_k is larger than $\delta_0^{2^k}$ only by a constant factor which is independent of the initial approximation error.

Based on Theorem 32 and Equation C.10 the error bound in setting II satisfies:

$$\rho(N_k') < (2k+1) \cdot \hat{n} + e^{\alpha} \cdot (|e_0| + 3d_0/2))^{2^k}.$$

C.4.3 Setting III: $n_i, d_i \leq \hat{n}$ and f_i/δ_i^2 is constant.

In setting III it is assumed that $f_i/\delta_i^2 \leq C$, where C is an appropriately chosen constant which is independent of the number of iterations k. In addition, it is assumed that $n_i, d_i \leq \hat{n}$, for every i.

Claim 35

$$\delta_k \;\leq\; (1+C)^{2^k-1} \cdot \delta_0^{2^k}.$$

Based on Theorem 32 and Equation C.10 the error bound in setting III satisfies:

$$\rho(N_k') < (2k+1) \cdot \hat{n} + ((1+C) \cdot (|e_0| + 3d_0/2))^{2^k}.$$

C.4.4 Setting IV: $n_i, d_i \leq \hat{n}$ and $f_i \leq \hat{f}$ for every i.

In setting IV the assumptions are: (i) $n_i, d_i \leq \hat{n}$, for every i, and (ii) $f_i \leq \hat{f} \leq 1/8$, for every i. Hence, $\delta_i \leq \delta_{i-1}^2 + \hat{f}$ for all $i \geq 0$.

The following claim bounds the error term δ_k corresponding to the kth iteration of Algorithm 4.

Claim 36 *Let $\alpha = (1 + \sqrt{\hat{f}})$. For every $k > 0$ the following holds:*

$$\delta_k \;\leq\; \hat{f} \;+\; \max\{\alpha^{2^{k+1}-2} \cdot \delta_0^{2^k} \;,\; (\alpha^{2^k-2} \cdot \delta_0^{2^{k-1}} + \hat{f})^2 \;,\; 9\hat{f}^2\}$$

Proof: We choose the substitution $\zeta_i = \delta_i - \hat{f}$, for $i > 0$, and $\zeta_0 = \delta_0$. Then, for $i > 0$:

$$\begin{aligned} |\zeta_i| &= |\delta_i - \hat{f}| \\ &\leq \delta_{i-1}^2 \\ &= (\zeta_{i-1} + \hat{f})^2 \end{aligned} \tag{C.11}$$

Observe that if $|\zeta_{i-1}| \leq \sqrt{\hat{f}}$, then $|\zeta_i| \leq 2 \cdot \hat{f} \leq \sqrt{\hat{f}}$. This is proved as follows. Since $\hat{f} \leq 1/4$, it follows that $2\hat{f} \leq \sqrt{\hat{f}}$. By Eq. C.11 we get:

$$\begin{aligned} |\zeta_i| &\leq (|\zeta_{i-1}| + \hat{f})^2 \\ &\leq (\sqrt{\hat{f}} + \hat{f})^2 \\ &\leq 2 \cdot \hat{f}. \end{aligned}$$

The last line holds since $\hat{f} \leq (\sqrt{2} - 1)^2 \approx 0.171$.

Let ℓ denote the highest index such that $\zeta_0, \ldots, \zeta_{\ell-1} > \sqrt{\hat{f}}$. Applying Equation C.11 we get for every $i \leq \ell$:

$$\begin{aligned} |\zeta_i| &\leq (|\zeta_{i-1}| + \hat{f})^2 \\ &\leq (|\zeta_{i-1}| + \sqrt{\hat{f}} \cdot |\zeta_{i-1}|)^2 \\ &= (\alpha \cdot \zeta_{i-1})^2. \end{aligned}$$

Hence by induction:

$$|\zeta_\ell| \leq \alpha^{2^{\ell+1}-2} \cdot \zeta_0^{2^\ell}, \tag{C.12}$$

If $\ell \geq k$, then the claim holds. Otherwise, there are two possibilities: (i) $\ell \leq k - 2$ or (ii) $\ell = k - 1$. If case (i) holds, then $\zeta_\ell \leq \sqrt{\hat{f}}$, and hence $|\zeta_j| \leq 2 \cdot \hat{f}$, for every $j > \ell$. Applying

Equation C.11 we get:

$$\begin{aligned} |\zeta_k| &\leq (|\zeta_{k-1}| + \hat{f})^2 \\ &\leq 9\hat{f}^2. \end{aligned}$$

If case (ii) holds then by applying Equation C.11 we get:

$$\begin{aligned} |\zeta_k| &\leq (|\zeta_\ell| + \hat{f})^2 \\ &\leq (\alpha^{2^k-2} \cdot \delta_0^{2^{k-1}} + \hat{f})^2 \quad \text{(by Eq. C.12).} \end{aligned}$$

Combining these cases (namely: $\ell \geq k, \ell = k-1$ and $\ell < k-1$) yields:

$$\zeta_k \leq \max\{\alpha^{2^{k+1}-2} \cdot \delta_0^{2^k} \; , \; (\alpha^{2^k-2} \cdot \delta_0^{2^{k-1}} + \hat{f})^2 \; , \; 9\hat{f}^2\}.$$

and the claim follows. □

Note that a slightly looser bound that does not involve a max function can be written as:

$$\delta_k \leq \hat{f} + \alpha^{2^{k+1}-2} \cdot \delta_0^{2^k} + 7\hat{f}^{3/2}.$$

Based on Theorem 32 and Equation C.10 the error bound in setting IV satisfies:

$$\rho(N_k') < (2k+1) \cdot \hat{n} + \hat{f} + \max\{\alpha^{2^{k+1}-2} \cdot \delta_0^{2^k} \; , \; (\alpha^{2^k-2} \cdot \delta_0^{2^{k-1}} + \hat{f})^2 \; , \; 9\hat{f}^2\}.$$

One can easily see that, due to the first term, there is a threshold above which increasing the number of iterations (while maintaining all other parameters fixed) increases the bound on the relative error. Moreover, the contribution of the error term $\hat{f}$ to $\rho(N_k')$ does not increase with the number of iterations k (as opposed to $\hat{n}$). This implies that in a cost effective choice one would use $\hat{f} > \hat{n}$.

C.5 Application: An Alternative FP-DIV Micro-architecture for AMD-K7$^{\text{TM}}$

In this section we propose an alternative FP-DIV micro-architecture for the AMD-K7 microprocessor [28]. This alternative micro-architecture is a design that implements Algorithm 4. Our micro-architecture uses design choices that are similar to those of [28] to facilitate isolating the effect of precisions on cost. Our error analysis allows us to accurately determine the required multiplier precision and thus both save cost and reduce delay.

Overview micro-architecture. The FP-DIV micro-architecture of the AMD-K7 microprocessor is described in [28]. The micro-architecture is based on Goldschmidt's algorithm. We briefly outline this micro-architecture: (i) Round-to-nearest rounding is done in intermediate computations (as opposed to directed rounding suggested in Algorithm 4). (ii) The design contains a single 76 × 76-bits multiplier. This means that the absolute errors $\widehat{neps}_i$ and $\widehat{deps}_i$ are identical during all the iterations (i.e., since round-to-nearest is used, $\widehat{neps}_i = \widehat{deps}_i = 2^{-76}$). However, our alternative micro-architecture may use smaller multipliers (even multipliers in which the multiplicands do not have equal lengths) provided that the error analysis proves that the final result is accurate enough. (iii) Intermediate results are compressed and represented using non-redundant binary representation. This means that Assumption 30 on strict directed rounding is easy to implement in our alternative micro-architecture. (Recall that directed rounding is used in Algorithm 4.) (iv) The computation of F_i' is done using one's complement computation. This means that the absolute error $\widehat{feps}_i$ is identical during all the iterations, and that the error analysis of Setting IV is applicable for our alternative architecture. (v) Final rounding of the quotient is done by back multiplication. Our alternative micro-architecture uses the same final rounding simply by meeting the same error bounds needed in the final rounding of [28].

Required final precisions. The micro-architecture in [28] supports multiple precisions: single precision $(24, 8)$ in one iteration, double precision $(53, 11)$ in two iterations, an extended precision $(64, 15)$ and an internal extended precision $(68, 18)$ in three iterations. Final rounding is based on back-multiplication: namely, comparing $N'_k \cdot B$ with A. In general, correct IEEE rounding based on back-multiplication requires that $\rho(N'_k) < 2^{-(p+1)}$, where p denotes the precision. (The description of the required precision for correct rounding in [28] is somewhat confusing since it is stated in terms of a two sided absolute error. For example, the absolute error in the 68-bit precision is bounded by 2^{-70}.)

To summarize, the upper bounds on the relative errors are as follows: (i) for single precision: $\rho(N'_1) < 2^{-25}$, (ii) for double precision: $\rho(N'_2) < 2^{-54}$, (iii) for extended double precision: $\rho(N'_3) < 2^{-65}$, and (iv) for the 68-bit precision: $\rho(N'_3) < 2^{-69}$.

Note that the bound for the 64-bit precision is weaker than the bound for the 68-bit precision. The bound for single precision is easily satisfied by the parameter choices needed to satisfy the 53-bit precision. Hence we focus below on two iterations for double precision and on three iterations for the 68-bit precision.

From relative errors to multiplier dimensions. We first discuss the lengths of the multiplicands in Algorithm 4 since these lengths determine the cost and delay of the multiplier. The lengths (or precisions) of the multiplicands is derived from upper bounds on the relative errors n_i, d_i and f_i.

In Algorithm 4 the multiplier performs the multiplications: $N'_i \cdot F'_i$ and $D'_i \cdot F'_i$. It is reasonable to assume that one multiplicand is used for N'_i and D'_i and that the other multiplicand is used for F'_i. (We later elaborate on which multiplicand should be Booth recoded.)

For simplicity, let us assume that Setting IV holds (i.e., $n_i, d_i \leq \hat{n}$ and $f_i \leq \hat{f}$, for every i).

We start by deriving the required length of the first multiplicand (used for N'_i and D'_i). Let a denote the length of the first multiplicand; this means that this multiplicand is represented by a binary string with bits in positions $[0 : (a - 1)]$. Since the product is rounded (either down or up) to length a, the absolute errors satisfy:

$$|\widehat{neps}_i|, |\widehat{deps}_i| < 2^{-(a-1)}.$$

Let us first focus on the constraint $n_i \leq \hat{n}$. By Claim 33 it follows that for $i \geq 2$:

$$0 \leq n_i \leq \frac{2\widehat{neps}_i}{1 - \pi_{i-1} - \delta_{i-1}} < \frac{4 \cdot 2^{-a}}{1 - \pi_{i-1} - \delta_{i-1}}.$$

Hence it suffices for a to satisfy:

$$\frac{4 \cdot 2^{-a}}{1 - \pi_{i-1} - \delta_{i-1}} \leq \hat{n}.$$

Which implies

$$a > \log_2 \frac{1}{\hat{n}} + 2 + \log_2 \frac{1}{1 - \pi_{i-1} - \delta_{i-1}}.$$

Similar lower bounds for a are derived for the cases $i = 0, 1$ using Claim 37. Hence a should be slightly larger than $\log_2(1/\hat{n}) + 2$.

Applying Claim 33 with respect to d_i yields that D_i' needs to be represented using slightly more than $1 + \log_2(1/\hat{n})$ bits. Hence, supporting the relative error n_i requires an extra bit compared to d_i.

We remark that this extra bit stems from the fact that N_i' can be less than 1. One could apply normalization shifts to the binary string that represents N_i' to reduce, in effect, the absolute error when $N_i' < 1$. This implies a reduction of the length a of the first multiplicand by 1 at the cost of adding the circuitry required for the normalization shift. Moreover, the normalization shift can take place in the 2nd pipeline stage (which has some slack with respect to delay) and help reduce the delay of the 3rd pipeline stage (which contains the addition tree of the multiplication tree). We do not take this optimization method into account in our comparison since it does not follow directly from our error analysis (which is the main contribution of this paper).

Applying Claim 33 and Claim 37 with respect to f_i yields (due to $i = 0$) that the length of the second multiplicand should be greater than or equal to $log_2(1/\hat{f}) + 1 + \log_2(1/(1 - \delta_0))$.

Optimizing the error parameters. Given a relative error bound on $\rho(N'_k)$, a combination of relative errors for intermediate computations is *feasible* if our error analysis shows that the relative error $\rho(N'_k)$ is smaller than the required bound. In this paragraph, we search for feasible combinations that minimize the sizes of the multiplier and lookup table. We used a cost function that is based on the cost model of Paul & Seidel [29] for Booth Radix-8 multipliers and the cost model of Paul & Mueller [25] for lookup tables, adders, etc. (Formulas for hardware costs appear in Section C.7.)

We demonstrate the power of our error analysis in finding optimal error parameters using Setting IV. Three error parameters determine the relative error $\rho(N'_k)$ in Setting IV: $\hat{n}$, $\hat{f}$ and e_0. Figure C.4 depicts the results of a three dimensional search for the triple $(\hat{n}, \hat{f}, e_0)$ that leads to the cheapest design that supports IEEE double precision (53-bits). Feasible pairs $(\hat{n}, \hat{f})$ that support double precision division are depicted in the colored region above the curve in Figure C.4. Each point in this region is assigned a maximum value of e_0, so that the triple $(\hat{n}, \hat{f}, e_0)$ is a feasible combination of error parameters. The values of $\hat{n}$ and $\hat{f}$ determine the multiplier dimensions (and hence its cost), and the value of e_0 determines the size of the lookup table (and hence its cost). In this fashion we are able to associate a cost to every feasible pair $(\hat{n}, \hat{f})$ (we used here a smooth continuous version of the cost function so that tradeoffs are easier to interpret). Closed curves are used to connect parameter combinations that lead to designs with equal cost. The parameter combination corresponding to the cheapest double precision design is depicted in Figure C.4. The optimal parameter combination is $-\log_2(\hat{f}) = 55.67$, $-\log_2(\hat{n}) = 57.74$, and $-\log_2(e_0) = 13.92$.

To simplify the search for the 68-bit internal precision we proceeded as follows. Setting I was employed to compute a good choice for e_0. Setting I assumes not only a bound $\hat{n}$ for every n_i and d_i; it also assumes that $f_i = 0$. We note, however, that following [28] our micro-architecture uses one's complement subtraction, so $f_i \neq 0$[1]. Nevertheless, Setting I is useful for the purpose of evaluating e_0. Using Setting I, we compute feasible pairs $(\hat{n}, e_0)$, so that the relative errors satisfy

[1] One may use injections in the micro-architecture to account for the missing ULP due to one's complement subtraction. However, we do not pursue this correction because we wish to stick to a micro-architecture as similar as possible to that in [28].

$\rho(N_3') < 2^{-68}$ and $\rho(N_2') < 2^{-53}$. Figure C.3 depicts the curves of feasible pairs $(\hat{n}, e_0)$ for double precision and internal 68-bit precision, respectively. The area above these curves constitutes the feasible area per constraint, hence we are interested in the intersection of the areas above the curves. We conclude that the pair $\hat{n} = 2^{-70.81}$ $e_0 = 2^{-13.51}$ is a feasible pair for all the required precisions. It is also evident that it is of no use to try to increase e_0 above $2^{-13.51}$. (We do need to decrease $\hat{n}$ due to the fact that $\hat{f} \neq 0$).

Based on the analysis above, we fix $e_0 = 2^{-13.51}$, and search for an optimal feasible pair $(\hat{n}, \hat{f})$ for the 68-bit precision. Such a feasible pair is also a feasible pair for the remaining precisions. The top figure in Figure C.5 depicts feasible pairs $(\hat{n}, \hat{f})$ for 68-bit precision. Recall that the length of the first multiplicand length is slightly more than $2+\log_2(1/\hat{n})$, and that the length of the second multiplicand is slightly more than $1 + \log_2(1/\hat{f})$. One can show that this small quantity that must be added when determining the multiplicands' lengths is less than 0.09. The bottom figure depicts the cost of the micro-architecture as a function of $\hat{f}$ (when $\hat{n}$ is taken to be as large as possible). The cost function took into account the discretization due to computing the multiplicands' lengths. Multiplier dimensions are depicted in the bottom figure. Note that either multiplicand could be Booth recoded (we use radix-8 as in [28]). Each instance of multiplier dimensions (e.g., 70 × 74) means that the first multiplicand A is preprocessed to compute $3A$, and the second multiplicand is Booth recoded. This explains the small effect that swapping the multiplicands has on the cost (e.g., 70 × 74 vs. 74 × 70). The minimum cost is obtained for the feasible pair $-\log_2 \hat{n} = 71.91$ and $-\log_2 \hat{f} = 68.9$. We conclude that multiplier dimensions 70 × 74 combined with a relative error bound $e_0 \leq 2^{-13.51}$ are a feasible choice of error parameters[2]. These parameters lead to a savings in cost of 10.6% compared to the micro-architecture described in [28].

[2] If one considers delay (rather than cost), then multiplier dimensions 74 × 70 lead to a slightly faster design (although slightly more expensive). Reduction in delay is due to the number of Booth digits: 24 Booth digits (corresponding to a 70-bit multiplicand) vs. 25 Booth digits (corresponding to a 74-bit multiplicand). Note that in [28] there are 26 Booth digits.

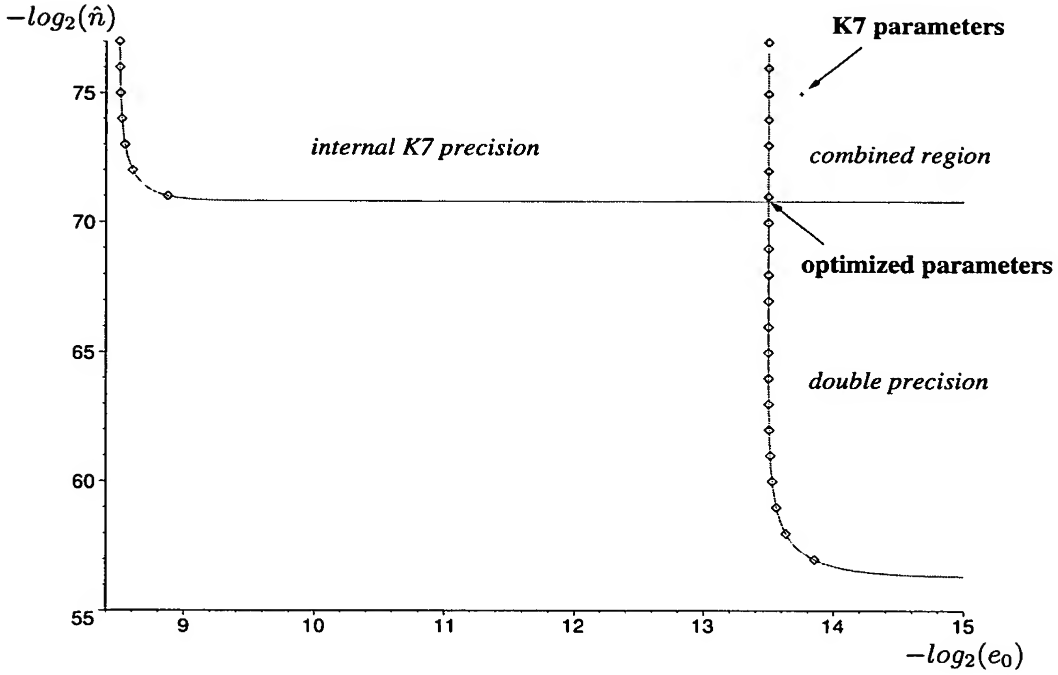

Figure C.3: Feasible parameter combinations of e_0 and $\hat{n}$ for double precision and extended double precision based on Setting I.

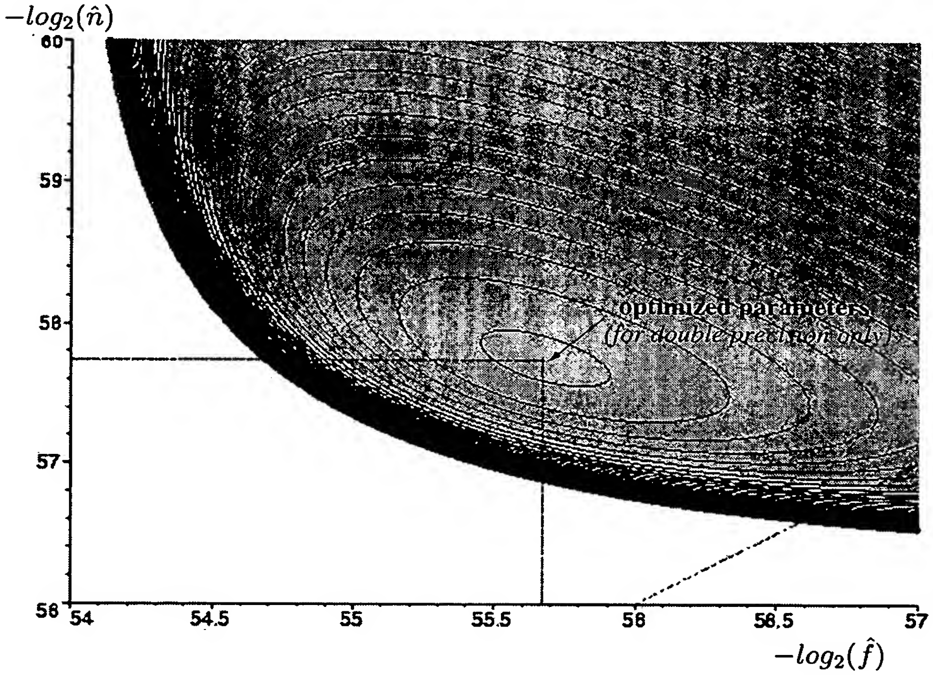

Figure C.4: Feasible $(\hat{n}, \hat{f})$ pairs using Setting IV for double precision are located in the shaded region above the curve. The closed curves depict pairs that lead to designs with equal costs. The central point depicts a feasible pair that leads to the cheapest design.

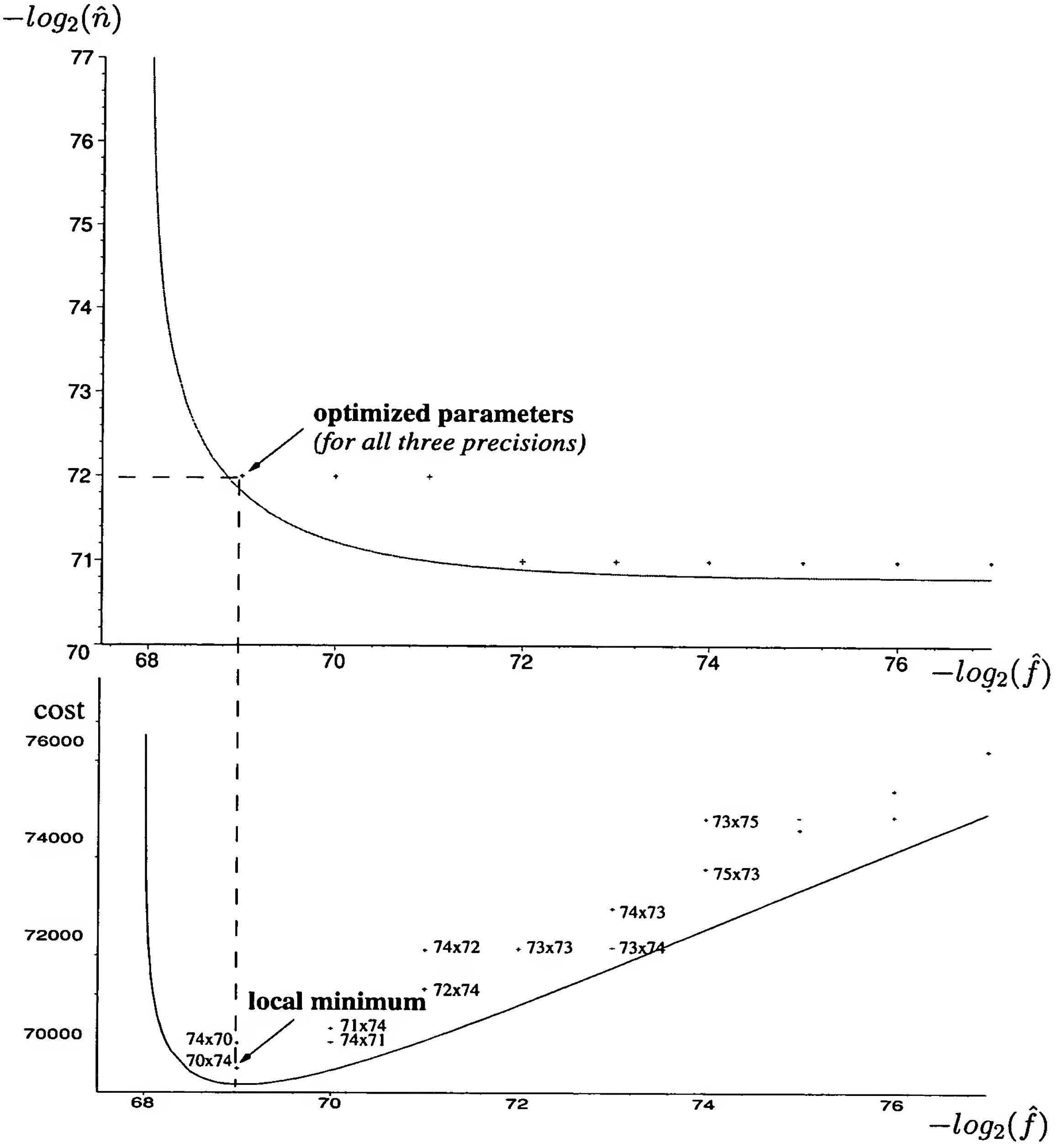

Figure C.5: Top: Feasible $(\hat{n}, \hat{f})$ pairs using setting IV for 68-bit precision (when $e_0 = 2^{-13.51}$). Bottom: Cost of design as a function of $\hat{f}$.

C.6 Bounds on relative error for first two iterations

Claim 37

$$
\begin{aligned}
n_0 &= \frac{neps_0}{(1-e_0)\cdot A/B} \leq \frac{2\cdot neps_0}{1-e_0}\\
d_0 &= \frac{deps_0}{(1-e_0)}\\
f_0 &\leq \frac{feps_0}{(1-\delta_0)}\\
n_1 &\leq \frac{neps_1}{A/B\cdot(1-\pi_0-\delta_0)\cdot\frac{1}{2}\cdot(1-f_0)} \leq \frac{neps_1}{(1-f_0)\cdot(1-\pi_0-\delta_0)}\\
d_1 &\leq \frac{deps_1}{(1-f_0)\cdot(1-\delta_0^2)}\\
f_1 &\leq feps_1
\end{aligned}
$$

C.7 Cost model equations

In this section we present the formulas that are used in the cost model. These formulas are on the cost models described in [25, 29]. Implementation of recoding and selection logic for Booth recoding radix-8 are from [4]. The cost formulas are listed below:

$$
\begin{aligned}
cCLA(n) &:= 24\cdot\lceil n\rceil - 12\cdot\lceil log_2(\lceil n\rceil)\rceil + 6\\
caddertree(n,t,\delta) &:= (n-\delta)\cdot(t-2) + (\lceil log_2(t)\rceil - 2)\cdot t\cdot\delta/2\\
cBooth8tree(n,m) &:= 14\cdot addertree(\lceil n+8\rceil, \lceil\frac{m+1}{3}\rceil, 3)\\
cBooth8rec(n,m) &:= 22\cdot\lceil\frac{m+1}{3}\rceil\\
cBooth8sel(n,m) &:= 18\cdot\lceil n+8\rceil\cdot\lceil\frac{m+1}{3}\rceil\\
cBooth8(n,m) &:= cCLA(n+2) + cBooth8tree(n,m) + cBooth8rec(n,m) +\\
&\quad + cBooth8sel(n,m) + cCLA(n+m)\\
Crom(A,d) &:= 0.25\cdot(A+3)(d + log_2(log_2(d)))\\
TLU(bit) &:= Crom(2^{bit}, bit-1)
\end{aligned}
$$

Appendix D

Deeply pipelined multiplicative division with IEEE rounding using a full size multiplier with redundant feedback

(A preliminary version of this paper appeared in [12])

Abstract

We present a pipelined implementation of Goldschmidt's division algorithm with IEEE rounding. The core of the design is a Booth radix-8 multiplier, the operands of which are slightly longer than the target precision (i.e., a "full-size" multiplier). The pipeline has four stages: (i) initial approximation, (ii) computation of the $3\times$ multiple of one operand and Booth recoding of the second operand, (iii) addition of the partial products to derive a carry-save representation of the product, and (iv) 2:1-addition and rounding selection.

If the precision of the initial is roughly $p/2$, where p denotes the target precision, then the latency is 9 cycles and a new division can be started after 6 cycles. This setting is especially suited for single precision (i.e., $p = 24$). In the case of double precision (i.e., $p_d = 53$), if the precision

of the initial is roughly $p_d/4$, the latency is 11 cycles and the restart-time is 8 cycles.

Our implementation uses several techniques to reduce the latency, among them: (i) Both operands of the Booth multiplier can be either represented as non-redundant binary-numbers or redundant carry-save numbers. (ii) A novel rounding procedure, called dewpoint rounding, is introduced. Dewpoint rounding simplifies the rounding decision, unifies rounding for all rounding modes, and avoids using tables. (iii) Injection based rounding is used to implement directed rounding of intermediate results.

The paper is self-contained and includes a detailed and complete description for single precision division. A simple error analysis is presented from which the exact multiplier dimensions and precision of the initial approximation are derived.

D.1 Introduction

Goldschmidt's division algorithm was developed in the early 60's to overcome the problem of two dependent multiplications per iteration in Newton's method. The advantage of two independent multiplications per iteration is that these two multiplications can be parallelized or pipelined to reduce latency. Although Goldschmidt's algorithm requires fewer cycles compared to Newton's method, very few designs are based on Goldschmidt's algorithm. The main reason that designers avoided Goldschmidt's algorithm is that it is not self-correcting. Namely, inaccuracies in intermediate computations accumulate and cause the computed result to drift away from the accurate quotient.

Recently, a general and parametric error analysis of Goldschmidt's algorithm was presented [13]. This error analysis allows different one-sided errors in each iteration and deals with an arbitrary number of iterations. In [13], the implementation of Goldschmidt's algorithm in AMD's K7 floating-point microarchitecture [28] was revisited using this parametric error analysis. It was shown that AMD's K7 floating-point microarchitecture is quite conservative. Namely, the same IEEE rounded quotient can be computed with a smaller multiplier (i.e., shorter operands) and less precise initial approximation (i.e., smaller tables).

In this paper we address the issue of further improving the floating-point microarchitectures that implement Goldschmidt's algorithm. For a target precision of p bits, we focus on a microarchitecture with the following characteristics:

1. The core of the microarchitecture is a "full-size" multiplier, namely, the dimensions of the multiplier are $(p+\varepsilon_1) \times (p+\varepsilon_2)$. The multiplier is a Booth radix-8 that is split into 3 pipeline stages. Such a pipeline allows for short clock cycles.
2. The precision of the initial approximation is roughly $p/2$ bits. Since the number of bits of precision roughly doubles per iteration of Goldschmidt's algorithm, only one iteration is needed to meet the target precision.

The exact multiplier dimensions and precision of the initial approximation are derived and minimized using a simple error analysis provided in this paper. We present an implementation of Goldschmidt's algorithm with such a microarchitecture that has a latency of 9 cycles and enables starting a new division operation after 6 cycles.

In [12] a microarchitecture that integrates a single precision and a double precision floating-point divider is outlined. To reduce cost and the clock period, the core of the design in [12] for double precision division is a half-size multiplier (i.e., somewhat larger than a $p_d + \epsilon_1 \times (p_d/2) + \epsilon_2$ multiplier, where $p_d = 53$). Two iterations of Goldschmidt's algorithm suffice for double precision since the precision of the initial approximation is roughly $p_d/4$. Such a microarchitecture constitutes, with respect to single precision (i.e., $p_s = 24$), a microarchitecture with a full-size multiplier and the precision of the initial approximation is at least $p_s/2$. Hence, this paper provides a complete and self-contained description of the microarchitecture for the case of single precision.

Related work. Many alternative approaches have been considered for floating-point division, some of these approaches are based on non-multiplicative techniques [27, 24]. Comparisons between multiplicative and non-multiplicative techniques are rare. Such a comparison appears, however, in [24], where various non-multiplicative methods are compared with a multiplicative imple-

design	latency (restart time) single precision	latency (restart time) double precision
SRT, (basic radix-2)	75* (75)	280* (280)
SRT, (basic radix-4)	78* (78)	170* (170)
SRT, (basic radix-16)	56* (56)	120* (120)
SRT, (short reciprocal)	65*	90*
based on fpmult	104 (85)	160 (136)
VHR (very high radix)	44*	65*
Boost VHR	44*	65*
Proposed Implementation of Goldschmidt's Algorithm	45 (30)	66 (48)

Table D.1: Latency comparison on division algorithms for single and double precision. The delay unit is the delay of a full adder. The top rows follow the model and estimates from [24]; the bottom row is the latency of the the proposed design. Latency estimates marked above with * require an additional delay for the IEEE rounding decision and selection and the implementation of all four IEEE rounding modes.

mentation based on Newton-Raphson's method. Delays are expressed in [24] in terms of the delay of a full adder, and the timing model does not consider routing delays. However, these timing estimates enable a technology independent comparison. We extend the timing estimates provided by [24] for double precision to single precision as well. These timing estimates appear in Table D.1 together with an estimate of the delay of the implementation of Goldschmidt's algorithm presented in this paper.

The estimate of the delay of our implementation is based on assuming that the clock period equals the delay of 5 full-adders in single precision and 6 full-adders in double precision. Since the latency is 9 and 11 cycles for single and double precision, respectively, the total delay is 45 and 66 full-adders, respectively. Although this timing analysis is very rough (since it ignores wire delays and gate sizing), it indicates that the proposed implementation of Goldschmidt's division algorithm is rather competitive with the best floating point division algorithms developed to date.

Techniques. The analysis presented in [13] assumes that all intermediate products are represented in a non-redundant representation. To reduce latency, our microarchitecture permits carry-save representation of the multiplication operands. We therefore begin with a simple and short error analysis for one iteration of Goldschmidt's algorithm (see Sec. D.3).

We present a novel rounding procedure for IEEE floating point division (see Sec. D.4). We refer to this rounding procedure as *dewpoint rounding*. The procedure relies on an error range of the quotient that allows for only two candidates for the final IEEE rounded result. We associate with each candidate r a rounding interval I_r. The rounding interval I_r is simply the set of numbers that are rounded to r. The number that separates the two rounding intervals is called the *dewpoint*. The rounding decision is obtained by comparing the dewpoint and the exact quotient by applying back-multiplication. We present a unified dewpoint rounding procedure for all rounding modes and avoid rounding tables. Previous approaches for IEEE rounding in division compute a rounding representative of the exact quotient (i.e., a number that belongs to the same rounding interval that the exact quotient belongs to) and the round the representative [34].

We employ additional techniques, among them: (i) A Booth recoded multiplier that can be fed by either non-redundant binary operands or by redundant carry-save operands. This technique when applied to a Booth radix-8 multiplier enables reducing the feedback latency to two cycles. Previously, Booth multipliers with one operand in redundant carry-save representation were studied [40, 8, 35]. (ii) Injection based rounding is used to implement directed rounding of intermediate results [9]. (iii) The rounding decision is based on the sign of the difference between the dewpoint and the exact quotient. We simplify the back-multiplication and the comparison by utilizing the fact that the difference between the dewpoint and the exact quotient has a small absolute value.

Organization. In Section D.2, notation is introduced and the version of Goldschmidt's algorithm from [13] is reviewed. In Section D.3, an error analysis of this division algorithm appears. In Section D.4, dewpoint rounding is presented. In Section D.5, we list various techniques employed to save cost and delay in a hardware implementation. In Section D.6, we present an implementation of the division algorithm that is based on a full-size multiplier. For the sake of concreteness, the presentation is for single precision division.

D.2 Preliminaries

Notation Let $x_i x_{i+1} \cdots x_j \in \{0,1\}^*$ denote a binary string. We often denote this string also by $x[i : j]$. We also refer to x_i as $x[i]$. Since we deal with fractions (mostly in the binade $[1, 2)$), the weight associated with the bit x_i is 2^{-i}. Namely, a fraction is represented by the binary digit string $x_0.x_1x_2 \cdots x_{p-1}$.

Let $\sigma = \sigma_i \sigma_{i+1} \cdots \sigma_j \in \{0,1,2\}^*$ denote a carry-save encoded digit string (i.e. $\sigma_i \in \{0,1,2\}$). A carry-save encoded digit string $\sigma[i : j]$ is represented by two binary vectors $\sigma'[i : j]$ and $\sigma''[i : j]$. Each carry-save digit $\sigma[\ell]$ satisfies $\sigma[\ell] = \sigma'[\ell] + \sigma''[\ell]$.

Given a binary string $x[i, j]$ and a carry-save encoded digit string $\sigma[i : j]$, we denote the values represented by these string by $|x[i : j]|$ and $|\sigma[i : j]|$, respectively.

Consider three binary vectors x, y, z. We denote 3:2-addition by $FA(x, y, z)$. Namely, $FA(x, y, z)$ means that the three binary vectors x, y, z are fed to a line of full-adders. The output of $FA(x, y, z)$ is two binary vectors s and c (i.e., a carry-save number) that satisfy $|x| + |y| + |z| = |s| + |c|$.

By *truncating* a carry-save encoded digit string $\sigma[i : j]$ after position q we simply mean chopping off the tail and leaving only $\sigma[i : q]$. We denote truncation of σ after position q by $\lfloor \sigma \rfloor_q$.

We often regard a carry-save encoded digit string σ also as two binary vectors σ' and σ''. Hence if σ is a carry-save encoded digit string and x is a binary vector, then $FA(\sigma, x)$ simply means $FA(\sigma', \sigma'', x)$.

Division Operation We consider the following task which captures the main difficulty in IEEE floating-point division. The dividend and the divisor are represented by p-bit binary strings $A[0 : p-1]$ and $B[0 : p-1]$, where $|A|, |B| \in [1, 2)$. Our goal is to compute the binary string $Q[0 : p-1]$ that is the binary encoding of the rounded value of the normalized quotient. More formally,

1. Let $A' = A$ if $|A| \geq |B|$, and $A' =$ leftshift(A) if $|A| < |B|$. This way, the quotient $|A'|/|B|$ is normalized (i.e., in the binade $[1, 2)$).
2. Let $q = |A'|/|B|$.

3. Round q (according to the IEEE standard). The binary string $Q[0 : p-1]$ should satisfy $|Q| = rnd(q)$.

In this paper we consider single precision, namely $p = 24$. Four rounding modes are defined in the IEEE standard: round towards zero round towards zero (RZ), round to nearest even (RNE), round towards infinity (RI) and round towards minus infinity(RMI).

Goldschmidt's division algorithm In [13] a variation of Goldschmidt's division algorithm based on directed rounding is presented and analyzed. The algorithm is listed below as Algorithm 5.

Algorithm 5 Goldschmidt-Approx-Divide(A', B) - Goldschmidt's division algorithm using approximate arithmetic

```
Initialize: $N'_{-1} \leftarrow A'$, $D'_{-1} \leftarrow B$, $F'_{-1} \leftarrow \frac{1-e_0}{B}$.
for $i = 0$ to $k$ do
    $N'_i \leftarrow (1-n_i) \cdot N'_{i-1} \cdot F'_{i-1}$.
    $D'_i \leftarrow (1+d_i) \cdot D'_{i-1} \cdot F'_{i-1}$.
    $F'_i \leftarrow (1-f_i) \cdot (2-D'_i)$.
end for
Return($N'_i$)
```

Directed roundings are used for all intermediate calculations. For example, N'_i is obtained by rounding down the product $N'_{i-1} \cdot F'_{i-1}$. We denote by n_i the relative error incurred when $N'_{i-1} \cdot F'_{i-1}$ is rounded down. Rounding down translates to the assumption that $n_i \geq 0$. Similarly, rounding down is used for computing F'_i and rounding up is used for computing D'_i.

In this paper we focus on an implementation of Algorithm 5 with $k = 1$ iterations. Namely, N'_1 is used as the estimate of the exact quotient for computing the rounded quotient.

D.3 Error analysis

In this section we prove bounds on the relative error of N'_1.

Definition 12 *The relative error of N_1' with respect to $\frac{|A'|}{|B|}$ is denoted by $\rho(N_1')$*

$$\rho(N_i') \triangleq \frac{|A'|/|B| - N_i'}{|A'|/|B|}$$

The relative errors n_0, n_1, d_0, f_0, e_0 are not known to the Algorithm 5. Instead, we are able to derive upper bounds on these relative errors (see Sec. D.6.3). Namely, $\hat{n}_0 \geq n_0, \hat{n}_1 \geq n_1, \hat{d}_0 \geq d_0, \hat{f}_0 \geq f_0$, and $\hat{e}_0 \geq |e_0|$.

The following claim summarizes the bounds on $\rho(N_1')$. Note that all the assumptions on the relative errors are very easy to satisfy.

Claim 38 *Let $\delta_0 = |\hat{e}_0| + \frac{3}{2} \cdot \hat{d}_0$. If*

$$\begin{array}{ccc} 0 \leq n_0 \leq \hat{n}_0 \leq 1 & 0 \leq n_1 \leq \hat{n}_1 \leq 1 & 0 \leq d_0 \leq \hat{d}_0 \leq 1 \\ 0 \leq f_0 \leq \hat{f}_0 \leq 1 & 0 \leq |e_0| \leq \hat{e}_0 \leq 1/2 & \delta_0 \leq 1, \end{array}$$

then

$$0 \leq \rho(N_1') \leq 1 - (1 - \hat{n}_1) \cdot (1 - \hat{n}_0) \cdot (1 - \hat{f}_0) \cdot (1 - \hat{e}_0^2 - \hat{d}_0 \cdot (1 + \hat{e}_0)^2)$$

Proof:

$$\begin{aligned} N_1' &= (1 - n_1) \cdot N_0 \cdot F_0 \\ &= (1 - n_1) \cdot (1 - n_0) \cdot |A'| \cdot \frac{1 - e_0}{|B|} \cdot (1 - f_0) \cdot (1 + e_0 - d_0 + e_0 d_0). \end{aligned} \tag{D.1}$$

To prove the lower bound we need to prove that $N_1' \leq |A'|/|B|$. Since

$$0 \leq (1 - n_1) \cdot (1 - n_0) \cdot (1 - f_0) \leq 1,$$

it suffices to show that

$$0 \leq (1 - e_0) \cdot (1 + e_0 - d_0 + e_0 d_0) \leq 1. \tag{D.2}$$

The assumption that $|e_0| \leq \hat{e}_0 \leq 1/2$ implies that $e_0 - d_0 + e_0 d_0 \geq -\hat{e}_0 - \frac{3}{2}\hat{d}_0 = -\delta_0$. The assumption on δ_0 and e_0 implies that the lower bound in Eq. D.2 follows. The upper bound in Eq. D.2 follows from the assumption that $d_0 \geq 0$ since

$$(1 - e_0) \cdot (1 + e_0 - d_0 + e_0 d_0) = 1 - \left(e_0^2 + d_0 \cdot (1 - e_0)^2\right) \leq 1. \tag{D.3}$$

We now prove the upper bound on $\rho(N_1')$. The assumptions imply that $(1 - n_0) \geq (1 - \hat{n}_0) \geq 0$, and the same holds for n_1 and f_0. By Eq. D.1, it suffices to prove that

$$(1 - e_0) \cdot (1 + e_0 - d_0 + e_0 d_0) \geq 1 - \hat{e}_0^2 - \hat{d}_0 \cdot (1 + \hat{e}_0)^2 \geq 0. \tag{D.4}$$

Equation D.4 follows from Eq. D.3 and the assumptions. The upper bound follows. □

D.4 IEEE rounding of the quotient

In this section we describe how IEEE rounding is done in our algorithm. We present a novel method called *dewpoint rounding*.

D.4.1 Background on IEEE rounding

The IEEE Floating-Point Standard 754 [17] requires the implementation of all four IEEE rounding modes for all basic arithmetic operations and for all precisions supported. The four IEEE rounding modes comprise of round towards zero (RZ), round to nearest even (RNE), round towards infinity (RI) and round towards minus infinity(RMI).

Back-multiplication is the common method for computing the IEEE rounded quotient. In this method, the approximate quotient is multiplied by the divisor B, and then this product is compared with the dividend A' [19, 34]. The rounded result is then computed to be within one ULP of the approximate quotient depending on the rounding mode, the sign, and the comparison.

We follow the method of reducing the four IEEE rounding modes to three rounding modes RZ, RI, and RNU (round to nearest up) based on the numbers' sign [32].

Observation 39 *The exact quotient $|A'|/|B|$ cannot be a midpoint between two representable numbers. Therefore RNU and RNE are equivalent rounding modes with respect to division.*

Based on the above reductions and Observation 39, we only need to consider the three rounding modes RZ, RI, and RNU.

Injection based rounding [9] was introduced to further simplify rounding. Injection based rounding reduces these three rounding modes to truncation. Loosely speaking, this reduction is obtained by adding an injection that only depends on the rounding mode. Dewpoint rounding also uses injections, as described in the next section.

D.4.2 Dewpoint Rounding

Overview

The concept of dewpoint rounding is inspired by the following concept of a dewpoint in meteorology: the dewpoint is the threshold temperature at which condensation of water in air occurs.

If the quotient's estimate is close enough to the accurate un-rounded quotient, then rounding only needs to select between two values. We denote by $RC_1 < RC_2$ the two candidate values for the IEEE rounded result. Note that $RC_2 = RC_1 + 2^{-p+1}$. For each rounding candidate RC_j, we denote by I_j the rounding interval that comprises the pre-image of RC_j with respect to rounding. Namely, I_j is the set of numbers that are rounded to RC_j. The definition of RZ, RNU, and RI rounding implies that since RC_1 and RC_2 are successive representable numbers, the rounding intervals I_1 and I_2 share an endpoint. This common endpoint is called the *dewpoint*. The rounded quotient is selected to be either RC_1 or RC_2 based on the sign of $|A'| - |B| \cdot \mathit{dewpoint}$.

Dewpoint rounding requires that there are only two rounding candidates. We are able to meet this requirement if the following assumption on N'_k holds.

Assumption 40 *The quotient's estimate N'_k satisfies:* $0 \leq \rho(N'_k) < 2^{-p}$.

Observe that our assumption is weaker than the conventional requirement of $0 \leq \rho(N'_k) < 2^{-p-2}$

Computation of rounding candidates and dewpoint

Given a carry-save encoding $\Re'_k[0:t]$ of N'_k, we wish to compute the dewpoint and the rounding candidates RC_1 and RC_2. A unified computation method based on injections is used for all rounding modes. We begin by dealing with RZ rounding and then reduce the two other rounding modes RNU and RI to RZ.

For the sake of simplicity we refer to the binary representation of N'_k and denote it by $N'_k[0:t]$. Namely, $N'_k = |N'_k[0:t]| = |\Re'_k[0:t]|$. Note that full compression of N'_k is not required in the implementation. We use the binary representation just to simplify the description of the rounding. We use some notation related to the target precision p.

Definition 13 *We refer to multiples of* 2^{-p+1} *as* representable significands *and to odd multiples of* 2^{-p} *as* mid-points.

RZ rounding. Since $0 \leq \rho(N'_k) < 2^{-p}$ and $\frac{|A'|}{|B|} < 2$, it follows that $\frac{|A'|}{|B|} \in [N'_k, N'_k + 2^{-p+1})$. Let $\lfloor N'_k \rfloor_{p-1} \triangleq |N'_k[0:p-1]|$. Truncation of $N'_k[0:t]$ after position $p-1$ yields:

$$N'_k \in [\lfloor N'_k \rfloor_{p-1}, \lfloor N'_k \rfloor_{p-1} + 2^{-p+1}).$$

We conclude that

$$\frac{|A'|}{|B|} \in [\lfloor N'_k \rfloor_{p-1}, \lfloor N'_k \rfloor_{p-1} + 2 \cdot 2^{-p+1}). \tag{D.5}$$

In the RZ rounding mode the interval $[\lfloor N'_k \rfloor_{p-1}, \lfloor N'_k \rfloor_{p-1}+2\cdot 2^{-p+1})$ is the union of exactly two rounding intervals: $[\lfloor N'_k \rfloor_{p-1}, \lfloor N'_k \rfloor_{p-1}+2^{-p+1})$ and $[\lfloor N'_k \rfloor_{p-1}+2^{-p+1}, \lfloor N'_k \rfloor_{p-1}+2\cdot 2^{-p+1})$. Hence, the dewpoint in this case is $\lfloor N'_k \rfloor_{p-1}+2^{-p+1}$. The rounding candidates are simply $RC_1 = \lfloor N'_k \rfloor_{p-1}$ and $RC_2 = \lfloor N'_k \rfloor_{p-1} + 2^{-p+1}$.

This method fails for RI and RNU since in these rounding modes the interval $[\lfloor N'_k \rfloor_{p-1}, \lfloor N'_k \rfloor_{p-1} + 2 \cdot 2^{-p+1})$ intersects three rounding intervals.

Reduction of RI and RNU to RZ. We reduce the rounding modes RI and RNU to RZ by using injections [11] as follows.

Let $\text{INJ}_{rnd}(mode)$ denote an injection constant that depends only on the rounding mode, the target precision p, and the precision t of N'_k. We define $\text{INJ}_{rnd}(mode)$ as follows.

$$\text{INJ}_{rnd}(mode) \triangleq \begin{cases} 0 & \text{if } mode = RZ \\ 2^{-p} & \text{if } mode = RNU \\ 2^{-p+1} - 2^{-t} & \text{if } mode = RI. \end{cases} \tag{D.6}$$

For $x \in [1, 2)$, the addition of $\text{INJ}_{rnd}(mode)$ reduces RI and RNU to RZ in the following sense (c.f. [11]):

$$\begin{aligned} RNU(x) &= RZ(x + \text{INJ}_{rnd}(RNU)) \\ RI(x) &= RZ(x + \text{INJ}_{rnd}(RI)). \end{aligned} \tag{D.7}$$

Hence it suffices to compute $RZ(\frac{|A'|}{|B|} + \text{INJ}_{rnd}(mode))$.

RI & RNU Rounding. We now repeat the analysis for rounding modes RNU and RI as follows. Define N_{INJ} as follows:

$$N_{\text{INJ}} \triangleq N'_k + \text{INJ}_{rnd}(mode).$$

The following claim shows that the exact quotient is contained in one of two rounding intervals. We leave the proof as an easy exercise.

Claim 41

$$\frac{|A'|}{|B|} \in \begin{cases} [\lfloor N_{\text{INJ}} \rfloor_{p-1} - 2^{-p}, \lfloor N_{\text{INJ}} \rfloor_{p-1} + 3 \cdot 2^{-p}) & \textit{if mode = RNU} \\ (\lfloor N_{\text{INJ}} \rfloor_{p-1} - 2^{-p+1}, \lfloor N_{\text{INJ}} \rfloor_{p-1} + 2^{-p+1}) & \textit{if mode = RI} \end{cases}$$

Note that for rounding mode RZ we have $N_{\text{INJ}} = N'_k$ and the previous analysis for RZ is also covered by our current formalism.

The following claim summarizes dewpoint rounding for all three rounding modes.

Claim 42 *Let*

$$RC_1 \triangleq \lfloor N_{\text{INJ}} \rfloor_{p-1}$$

$$RC_2 \triangleq \lfloor N_{\text{INJ}} \rfloor_{p-1} + 2^{-p+1}$$

$$dewpoint \triangleq \begin{cases} RC_1 + 2^{-p+1} & \textit{if mode} = RZ \\ RC_1 + 2^{-p} & \textit{if mode} = RNU \\ RC_1 & \textit{if mode} = RI. \end{cases}$$

For the rounding modes mode $\in$ $\{RZ,RNU,RI\}$*, the IEEE rounded result has the following property:*

$$rnd_{mode}\left(\frac{|A'|}{|B|}\right) = \begin{cases} RC_1 & \textit{if} \left(\frac{|A'|}{|B|} < dewpoint\right) \\ RC_1 & \textit{if} \left(\frac{|A'|}{|B|} = dewpoint \textit{ AND } mode = RI\right), \\ RC_2 & \textit{otherwise.} \end{cases} \tag{D.8}$$

Note that the decision does not require division since $(\frac{|A'|}{|B|} < dewpoint)$ if and only if $|A'| < dewpoint \cdot |B|$.

In the description of the division algorithm a unified notation is used. For this purpose we introduce the dewpoint displacement constant $C_{dew}(mode)$ that is defined as as follows.

$$C_{dew}(mode) \triangleq \begin{cases} 2^{-(p-1)} & \text{if } mode = \text{RZ} \\ 2^{-p} & \text{if } mode = \text{RNU} \\ 0 & \text{if } mode = \text{RI}. \end{cases}$$

The *dewpoint* satisfies:

$$dewpoint \triangleq \lfloor N_{\text{INJ}} \rfloor_{p-1} + C_{dew}(mode).$$

D.5 Hardware Optimizations

D.5.1 Redundant Feedback & Partial Compression

Addition trees in multipliers are not amenable to pipelining. Short clock cycles are therefore not achievable with reasonable cost if the addition tree has too many rows. Booth radix-8 recoding reduces the number of rows in an addition tree from n to $\lceil \frac{n+1}{3} \rceil$.

Booth radix-8 multipliers are usually implemented using a 3-stage pipeline, as follows: (i) precompute the $3\times$ multiple of the first operand of the multiplier and recode the second operand, (ii) addition tree that computes a carry-save representation of the product, and (iii) final carry-propagate addition.

Goldschmidt's algorithm performs only 2 multiplications per iteration. Hence running Goldschmidt's algorithm on a 3-stage pipeline creates un-utilized cycles (i.e. "bubbles") in the pipeline. These bubbles increase the latency and reduce the throughput. The Athlon$^{\text{TM}}$ processor (in hardware) [28] and Itanium$^{\text{TM}}$ processor (in software) [5] attempt to utilize these bubbles (and increase throughput) by allowing other multiplications to be executed during such bubbles.

We suggest a Booth radix-8 multiplier that allows for both operands to be either in nonredundant representation or carry-save representation. Booth multipliers with one operand in redundant carry-save representation were studied in [40, 8, 35]. Allowing both operands to be in redundant representation conceptually reduces the 3-stage pipeline to a 2-stage pipeline for all but the last iteration of the algorithm.

The Booth-8 multiplier design that supports operands in redundant representation is not symmetric in the sense that the first operand and second operand of the multiplier are processed differently during the first pipeline stage. During the first pipeline stage, operands represented as carry-save numbers are processed as follows:

1. The first operand is compressed and its $3\times$ multiple is computed. This requires two adders. To save hardware, we suggest employing the adder from the third pipeline stage for the purpose of compressing the first operand. This explains why the compression of the first

operand appears in Table D.2 as an operation that takes place in the third pipeline stage.

2. The second operand can be partially compressed from carry-save representation before being fed to the Booth recoder. In Section D.7, we describe a recoding method that follows [35].

D.5.2 Directed Rounding of Carry-Save Numbers

We propose an implementation of Algorithm 5 in which intermediate results are represented by carry-save encoded digit strings. Since Algorithm 5 uses directed roundings for all intermediate computations, we need to explain how directed rounding is applied to carry-save numbers.

The following lemma deals with rounding up of a carry-save number. It shows that 3:2-addition followed by truncation after position q can be used to implement approximate rounding up. The 3:2-addition adds a constant, called the injection, that depends only on the rounding position and the length of the number.

Lemma 43 *Let $\sigma[0:t]$ denote a carry-save encoded digit string. Let $\text{INJCS}_{RU}(q,t) \in [2^{-q} + 2^{-(q+1)}, 2 \cdot (2^{-q} - 2^{-t})]$ and assume that $\text{INJCS}_{RU}(q,t)$ is represented by a binary string $I[q:t]$. Then,*

$$|\sigma[0:t]| \;\leq\; |\lfloor FA(\sigma[0:t], I[q:t]) \rfloor_q| \;\leq\; |\sigma[0:q]| + 2^{-q+1}.$$

Proof: We first show that

$$|\sigma[0:t]| \;\leq\; |\lfloor FA(\sigma[0:t], I[q:t]) \rfloor_q|. \tag{D.9}$$

Note that from the range of $\text{INJCS}_{RU}(q,t)$, it follows that $I[q] = 1$ and $I[q+1] = 1$. Hence,

$$\begin{aligned} |\lfloor FA(\sigma[0:t], I[q:t]) \rfloor_q| &= |\sigma[0:q]| + I[q] \cdot 2^{-q} + |\lfloor FA(\sigma[q+1:t], I[q+1:t]) \rfloor_q| \\ &= |\sigma[0:q]| + 2^{-q} + c_q \cdot 2^{-q}, \end{aligned} \tag{D.10}$$

where $c_q \in \{0,1\}$ is a carry bit to position q computed by the $3:2$-addition.

We now consider two cases:

1. $c_q = 1$: in this case

$$\begin{aligned}
|\lfloor FA(\sigma[0:t], I[q:t])\rfloor_q| &= |\sigma[0:q]| + 2^{-q+1} \\
&> |\sigma[0:t]|.
\end{aligned}$$

2. $c_q = 0$: since $I[q+1] = 1$, a carry is not generated to position $[q]$ only if $\sigma[q+1] = 0$. Hence the $|\sigma[q+1:t]| < 2^{-q}$, and therefore the lower bound in Eq. D.9 holds.

Since $c_q \leq 1$, the upper bound follows directly from Eq. D.10, and the claim follows. □

We present in Lemma 43 the widest possible interval for $\text{INJCS}_{RU}(q,t)$. It is probably most beneficial to use a value with the smallest number of ones, namely, $\text{INJCS}_{RU}(q,t) = 2^{-q} + 2^{-(q+1)}$.

D.5.3 Optimized Implementation of Dewpoint Rounding and Back Multiplication

In this section we present an optimized implementation for computing the z-sign (i.e. positive, negative, zero) of $dewpoint \cdot |B| - |A'|$. The z-sign of $dewpoint \cdot |B| - |A'|$ (combined with the rounding mode) determines whether the IEEE rounded quotient is RC_1 or RC_2.

The following claim shows that since the dewpoint is close to the exact quotient, the absolute value of $|B| \cdot dewpoint - |A'|$ is small.

Claim 44

$$-2^{-p+1} < |B| \cdot dewpoint - |A'| < 2^{-p+2}.$$

Proof: By Assumption 40, $0 \leq \rho(N_1') < 2^{-p}$. Recall that

$$\begin{aligned}
dewpoint &\triangleq \lfloor N_{\text{INJ}} \rfloor_{p-1} + C_{dew}(mode) \\
&= \lfloor N_k' + \text{INJ}_{rnd}(mode) \rfloor_{p-1} + C_{dew}.
\end{aligned}$$

It follows that

$$\begin{aligned} \mathit{dewpoint} &\le N'_k + \text{INJ}_{rnd}(\mathit{mode}) + C_{dew}(\mathit{mode}) \\ &\le N'_k + 2^{-p+1} \\ &\le \frac{|A'|}{|B|} + 2^{-p+1}. \end{aligned}$$

The second line follows from the definition of the rounding injection $\text{INJ}_{rnd}(\mathit{mode})$ and the dewpoint displacement constant $C_{dew}(\mathit{mode})$. The third line follows from $0 \le \rho(N'_1)$. The upper bound now follows since $B < 2$.

To prove the lower bound we need to show that $-\frac{2^{-p+1}}{|B|} + \frac{|A'|}{|B|} \le \mathit{dewpoint}$. Since $\rho(N'_1) < 2^{-p}$ and $\frac{|A'|}{|B|} < 2$, it follows that $\frac{|A'|}{|B|} < N'_1 + 2^{-p+1}$. Since $|B| \ge 1$ it suffices to prove that $N'_1 \le \mathit{dewpoint}$.

We consider each of the three rounding modes.

RZ: Let x and x' denote two successive representable significands that sandwich N'_1, namely, $N'_1 \in [x, x')$ where $x' = x + 2^{-p+1}$. In RZ, $\text{INJ}_{rnd}(RZ) = 0$ and $C_{dew}(RZ) = 2^{-p+1}$. Hence: $\lfloor N'_1 + \text{INJ}_{rnd}(\mathit{mode}) \rfloor_{p-1} = x$ and $\mathit{dewpoint} = x' > N'_1$.

RI: Assume here that $N'_1 \in (x, x']$ (i.e., the equality can hold in x' instead of x). In RI, $\text{INJ}_{rnd}(RI) = 2^{-p+1} - 2^{-t}$ and $C_{dew}(RNU) = 0$. Hence $\lfloor N'_1 + \text{INJ}_{rnd}(\mathit{mode}) \rfloor_{p-1} = x'$ and $\mathit{dewpoint} = x' \ge N'_1$.

RNU: Assume here that $N'_1 \in [x + 2^{-p}, x' + 2^{-p})$ (i.e., we sandwich N'_1 between two mid-points). In RNU, $\text{INJ}_{rnd}(RNU) = 2^{-p}$ and $C_{dew}(RNU) = 2^{-p}$. Hence $\lfloor N'_1 + \text{INJ}_{rnd}(\mathit{mode}) \rfloor_{p-1} = x'$ and $\mathit{dewpoint} = x' + 2^{-p} > N'_1$.

In all three rounding modes, $\mathit{dewpoint} \ge N'_1$, and hence the claim follows. □

We use the following notation: $\alpha = \mathit{dewpoint} \cdot |B| - |A'|$. Let $a[-2 : t]$ denote a two's complement non-redundant representation of α, namely:

$$\alpha = -a_{-2} \cdot 2^2 + \sum_{i=-1}^{t} a_i \cdot 2^{-i}.$$

The following claim suggests an optimized method for computing the z-sign of α.

Claim 45

$$\begin{aligned} \alpha < 0 &\iff a_{p-2} = 1. \\ \alpha = 0 &\iff a_{p-2} = 0 \text{ AND } (\text{OR}(a_{p-2}, \ldots, a_t) = 0), \\ \alpha > 0 &\iff a_{p-2} = 0 \text{ AND } (\text{OR}(a_{p-2}, \ldots, a_t) \neq 0), \end{aligned}$$

Proof: Let

$$\begin{aligned} a' &\triangleq -a_{-2} \cdot 2^2 + \sum_{i=-1}^{p-2} a_i \cdot 2^{-i} \\ a'' &\triangleq \sum_{i=p-1}^{t} a_i \cdot 2^{-i}. \end{aligned}$$

By definition $a'' \in [0, 2^{-p+2})$ and $a' \in (-\infty, -2^{-p+2}] \cup \{0\} \cup [2^{-p+2}, \infty)$. Hence, if $a' > 0$, then $\alpha \geq a' \geq 2^{-p+2}$, contradicting Claim 44. It follows that $a' \leq 0$.

We claim that $a' = 0$ or $a' = -2^{-p+2}$. Suppose that $a' < -2^{-p+2}$. Then $a' \leq -2 \cdot 2^{-p+2}$. In this case $\alpha = a' + a'' < -2 \cdot 2^{-p+2} + 2^{-p+2} = -2^{-p+2}$, a contradiction.

If $a' = -2^{-p+2}$, then $\alpha < 0$. If $a' = 0$, then $\alpha \geq 0$. Moreover, $\alpha = 0$ iff $a' = 0$ and $a'' = 0$. Similarly, $\alpha > 0$ iff $a' = 0$ and $a'' > 0$. □

We support negative numbers using a representation called two's complement carry-save representation [7]. In this representation, a number is represented by two binary vectors, each of which is a two's complement number (i.e., the most significant position has a negative weight).

In our implementation, a two's complement carry-save representation of α is computed. For the purpose of computing z-sign(α), we conclude that: (i) the carry-save digits of α in positions

$[p-2:t]$ suffice. (ii) the subtraction of $|A'|$ does not require using $4-A'$ as an addend; it suffices to use $2^{-(p-3)} - A'[p-2:p-1]$ (see cycle 7 in Table D.2).

D.6 An implementation with a full size multiplier

In this section we present an implementation of our algorithm that uses a full size multiplier (i.e., for a target precision p, the multiplier dimensions are roughly $p \times p$) and an initial approximation that whose precision is roughly $p/2$. To be concrete, we present detailed design for single precision (i.e. $p = 24$) that has a latency of 9 cycles.

The main motivation for this setting is an integrated single precision and double precision floating-point divider. We assume that core of the design for double precision division is a half-size multiplier (i.e., somewhat larger than a $p_d \times (p_d/2)$ multiplier, where $p_d = 53$). Such a multiplier is larger than a $2p \times p$ multiplier. This implies that, when performing single precision division, the multiplier is not only a full-size multiplier, but a double-width multiplier. We can employ a double-width multiplier to even reduce the latency from 9 cycles to 8 cycles (see Sec. D.6.4).

We note that an extension of the error analysis from Section D.3 to the case of double precision yields the following results. Correct IEEE rounding is achieved with a "full-size" multiplier (57×62 bits) and an initial approximation with precision $e_0 \leq 2^{-13.51}$. The corresponding schedule for such multiplier design is listed in Table B.3. We omit the proofs and details due to space limitations. This double precision design achieves a latency of 11 clock cycles (including the initial approximation and IEEE rounding).

D.6.1 Basic microarchitecture

A block diagram with the four stages of our basic microarchitecture is depicted in Fig. D.1. The core of this microarchitecture is a radix-8 Booth recoded multiplier. The microarchitecture allows for feeding of previously computed products back to the multiplier as either operands. Latency is reduced by allowing the feedback to be in redundant representation. The multiplier also supports a multiply-add operation (i.e. $x \cdot y + z$). The addend (i.e. the number to be added to the product) is a binary number, so only one row in the adder tree is allocated for the addend.

The multiplication circuitry is divided into 4 pipeline stages. Below we describe some details of the stages:

In pipeline stage 0, an estimate for the reciprocal $1/B$ is computed and the values of A and B are compared to determine the pre-normalized value A'.

In stage 1, the two operands of the multiplier (i.e., the multiplicand and multiplier) are prepared for the addition of the partial products in the second stage. The second operand (i.e., the multiplier) is recoded in Booth radix-8 digits and the partial products are generated. Following [35], the recoding can accept either a binary string or a carry-save encoded digit string.

The first operand (i.e., the multiplicand) is processed as follows. The $3\times$ multiple of the multiplicand is computed using an adder. A feedback product, encoded as a carry-save digit string, can be used as a multiplicand as follows. The computation of the $3\times$ multiple is preceded by a 4:2-adder that computes a carry-save encoding of the $3\times$ multiple. This carry-save encoded digit string is compressed to a binary number by the adder. Meanwhile, the binary representation of the multiplicand is computed by the adder in the third pipeline stage. This saves an adder in the first pipeline stage (we need to insure that the adder in the third pipeline stage is indeed available).

In stage 2, the partial products are compressed by an adder tree. In addition to the partial products, there is a row that is dedicated for either (i) a round-up rounding injection (ii) an IEEE rounding injection (see Equation D.6), or (iii) a two's complement number (see Section D.5.3).

Stage 3 contains an adder and a comparator. The adder has the following tasks: (i) compressing the multiplicands from carry-save representation to binary representation and (ii) computing RC_1 and RC_2.

The comparator is used for computing the sign of the back-multiplication. We cannot use the adder as a comparator due to conflicts between two pipelined divisions.

Additional circuitry is installed in the third stage, including: (i) circuitry for feeding the adder with 2^{-23} and (ii) circuitry for selecting between RC_1 and RC_2.

D.6.2 Pipeline schedule

Table D.2 lists the schedule of operations that implements our algorithm for single precision based on a "full-size" multiplier and a "half-size" initial approximation. In the error analysis presented in Section D.6.3 we show that a 30×27 rectangular multiplier together with an error bound of $|e_0| \leq 2^{-12.71}$ guarantees correctness. We use these parameters in the detailed description below. We now describe the cycles one by one.

cycle 0: The following computations are performed in cycle 0:

1. An initial approximation of $1/B$ is computed. We denote this approximate reciprocal by $(1 - e_0)/B$. Since this reciprocal is recoded in the next cycle, it is possible to have the reciprocal represented in binary or redundant format.

2. Comparison of A and B. The comparison determines A' as follows:

$$A'[-1:23] \leftarrow \begin{cases} \text{left-shift}(A[0:23]) & \text{if } A < B \\ A[0:23] & \text{otherwise.} \end{cases}$$

   Note that since $|A'|$ can be in the binade $[2, 4)$, a bit in position $[-1]$ is required.

cycle 1: Iteration 0 of the algorithm begins in this cycle. The approximate reciprocal $(1 - e_0)/B$ is assigned to F'_{-1} and is fed as the second operand of the multiplier. The approximate reciprocal is recoded as Booth radix-8 digit string. The first operand of the multiplier D'_{-1} is simply assigned the value B. The $3\times$ multiple of D'_{-1} is computed in this cycle.

cycle 2: Two activities take place in this cycle. In the first pipeline stage, $A'[-1:28]$ is assigned to N'_{-1} and the $3\times$ multiple of N'_{-1} is computed. In the second pipeline stage the rounded up product $D'_{-1} \cdot F'_{-1}$ is stored in $D'_0[0:28]$ as a carry-save number. The rounding up of a result represented in carry-save representation is done by using the rounding up injection particular to carry-save representations $\text{INJCS}_{RU}(28, 28+14)$ (see Sec. D.5.2).

cycle 3: The carry-save representation of D'_0 is compressed to binary representation in the third pipeline stage. In the second pipeline stage, the rounded down product $N'_{-1} \cdot F'_{-1}$ is stored in $N'_0[-1 : 28]$. Rounding down is simply achieved by truncation of the carry-save representation. Note that our error analysis (Claim 38) only implies that $N'_1 < 2$, hence we need position $[-1]$ in N'_0.

cycle 4: In the first pipeline stage, a binary representation of $F'_0 = 2 - 2^{-26} - D'_0$ is computed and then Booth recoded. (Recall that D'_0 is compressed to binary representation in the previous cycle.) In addition, the $3\times$ multiple of N'_0 is computed from the carry-save representation of N'_0.

In the third pipeline stage, the carry-save representation of N'_0 is compressed to binary.

cycle 5: In the second pipeline stage we compute the carry-save representation of N_{INJ}.

$$N_{\text{INJ}} \leftarrow N'_0 \cdot F'_0 + \text{INJ}_{rnd}(mode).$$

cycle 6: In the third pipeline stage, RC_1 is computed by compressing the carry-save representation of N_{INJ} and truncating at position 23. The dewpoint equals $RC_1 + c_{dew}(mode)$. However, to reduce delay, we compute the dewpoint as follows. Let $carry_{23}(N_{\text{INJ}}[24 : 28])$ denote the bit that indicates if $N_{\text{INJ}}[24 : 28] \geq 2^{-23}$. A carry-save representation of the dewpoint is obtained by adding $N_{\text{INJ}}[0 : 23] + c_{dew}(mode) + carry_{23}(N_{\text{INJ}}[24 : 28])$. The dewpoint is Booth recoded and the $3\times$ multiple of B is computed.

cycle 7: Back-multiplication takes place in the second pipeline stage. We use a representation called two's complement carry-save representation [7]. In this representation the most significant position has a negative weight. Our goal in the back multiplication is to compute the z-sign of

$\alpha = B \cdot dewpoint - A'$. (The z-sign indicates equality, greater than, or less than.) In Section D.5.3 we show that it suffices to examine digits in positions $[22 : 47]$ to determine the z-sign of α.

cycle 8: In the third pipeline stage, (i) RC_2 is computed, (ii) the z-sign of the back-multiplication is determined from $\alpha[22 : 47]$, and (iii) the IEEE rounded quotient is selected from RC_1 and RC_2 according to the rounding mode and the z-sign of α.

D.6.3 Correctness

In this section we prove the correctness of a design based on a rectangular 30 × 27-multiplier and an initial approximation with a relative error $|e_0| \leq 2^{-12.71}$. To support single precision ($p = 24$), dewpoint rounding requires that Assumption 40 holds, namely, $0 \leq \rho(N'_1) < 2^{-24}$.

The analysis presented in this section is a concrete analysis for single precision. More generally, this analysis applies to the following situation: (i) the accuracy of the initial approximation is roughly $2^{-p/2}$; (ii) the quotient is approximated by N'_1 (namely, the algorithm has 1 iteration of quadratic convergence); and (iii) the multiplier is a rectangular multiplier whose dimensions are not smaller than roughly $p \times p$.

In Table D.2 we assume that the initial approximation $F'_{-1} = \frac{1-e_0}{B}$ is represented by 15 bits. The following claim shows that a correct IEEE rounded quotient is computed.

Claim 46 *If the relative error in initial approximation satisfies* $|e_0| \leq 2^{-12.71}$, *then correct single precision IEEE rounding is obtained with a rectangular* 30×27*-bit multiplier.*

Proof: All we need to show is that $0 \leq \rho(N'_1) < 2^{-24}$. To use Claim 38 we prove upper bounds on the relative error terms (i.e., n_0, n_1, f_0, d_0). We bound these relative error terms using absolute error terms defined as follows:

Definition 14 *The absolute errors of intermediate computations are defined as follows:*

$$\begin{aligned} neps_i &\triangleq n_i \cdot N'_{i-1} \cdot F'_{i-1} \\ deps_i &\triangleq d_i \cdot D'_{i-1} \cdot F'_{i-1} \\ feps_i &\triangleq f_i \cdot (2 - D'_i). \end{aligned}$$

We now bound each of the relative error terms:

Bound on d_0: According to Table D.2, D'_0 is represented by a carry-save encoded digit string with digits in positions $[0 : 28]$. This digit string holds the *rounded up* value of $D'_{-1} \cdot F'_{-1}$. This implies that the absolute error $deps_0$ associated with the computation of D'_0 is in the range $[0, 2 \cdot 2^{-28})$ (see Lemma 43). Since the precise product $D'_{-1} \cdot F'_{-1}$ equals $1 - e_0$, this product is in the range $1 \pm 2^{-12.71}$. It follows that $d_0 < 2^{-27}/(1 - 2^{-12.71}) < 2^{-26.99978}$.

Bound on n_0: As in the case of D'_0, N'_0 is also represented by a carry-save encoded digit string with digits in positions $[0 : 28]$. This digit string holds the *rounded down* value of $N'_{-1} \cdot F'_{-1}$. This implies that the absolute error $neps_0$ associated with the computation of D'_0 is in the range $[0, 2 \cdot 2^{-28})$. The product $A' \cdot F'_{-1}$ equals $\frac{A'}{B} \cdot (1 - |e_0|)$. Since $A' \geq B$, it follows that $A' \cdot F'_{-1} \geq 1 - |e_0|$, and therefore, $n_0 < 2^{-27}/(1 - |e_0|)$.

Bound on f_0: Note that D'_0 is compressed to binary non-redundant representation in the third cycle. We first show that $feps_0 < 2^{-26}$. The reason is that instead of computing $2 - D'_0[0 : 28]$, we compute $2 - 2^{-26} - D'_0[0 : 26]$. The absolute error is $2^{-26} - D'_0[27 : 28]$. Hence, $feps_0 \leq 2^{-26}$, as required. Next, we expand $2 - D'_0 = 2 - (1 + d_0) \cdot (1 - e_0)$. We conclude that

$$\begin{aligned} f_0 &\leq \frac{2^{-26}}{2 - (1 + d_0) \cdot (1 + |e_0|)} \\ &< 2^{-25.999784}. \end{aligned}$$

Bound on n_1: As in the case of n_0, $neps_1 < 2^{-27}$.

We bound from below N'_0 and F'_0 as follows: $N'_0 = (1 - n_0) \cdot (1 - e_0) \cdot \frac{A'}{B}$. Hence, $N'_0 \geq (1 - n_0) \cdot (1 - |e_0|)$. Also, $F'_0 = (1 - f_0) \cdot (2 - D'_0) \geq (1 - f_0) \cdot (1 - \delta_0)$. We conclude that

$$\begin{aligned} n_1 &< \frac{2^{-27}}{(1 - n_0) \cdot (1 - |e_0|) \cdot (1 - f_0) \cdot (1 - \delta_0)} \\ &< 2^{-26.999569}. \end{aligned}$$

We now apply Claim 38 to obtain:

$$\begin{aligned} \rho(N'_1) &\leq 1 - (1 - \hat{n}_1) \cdot (1 - \hat{n}_0) \cdot (1 - \hat{f}_0) \cdot (1 - \hat{e}_0^2 - \hat{d}_0 \cdot (1 + \hat{e}_0)^2) \\ &\leq 2^{-24.0016} < 2^{-24}. \end{aligned}$$

□

D.6.4 Design alternatives

The proposed implementation is not the only way to implement our algorithm. Certain choices were made in order to keep the implementation simple without increasing cost by much. Below we list a few alternatives.

Latency reduction. One can execute cycles 2 and 3 simultaneously and reduce the latency from 9 cycles to 8 cycles. One way to execute the two multiplications simultaneously is to use a multiplier in which (i) the length of the first operand is wider than 29 + 30 bits and (ii) partial products corresponding to the two different products are not mixed. Note that such a multiplier is possible if double precision is supported (even if the multiplier is "half-size" with respect to double precision).

The two multiplications that are executed in cycles 2 and 3 have the same second operand (i.e., F'_{-1}). Hence a 60×15 multiplier with a possibility to separate the two type of partial products suffices. Note that compression of D'_0 during cycle 3 is not crucial and could be skipped at the expense of increasing the error term f_0.

Avoid the comparison $A < B$ and using A'. In this case the exact quotient (and hence N'_1) may belong to the binade $(1/2, 1)$. This means that N'_1 has to be normalized so that it is in the binade $[1, 2)$. This modification has the following implications: (i) The rounding injection cannot be added in cycle 5 since normalization changes the value of the injection. Hence in cycle 6 an injection (or injection correction) needs to be added when computing RC_1. (ii) The error analysis should be changed to deal with this case (slightly worse bounds will be obtained).

Avoid position $[-1]$ in the first operand of the multiplier. Position $[-1]$ is required in the first operand in two places: (a) in cycle 3 to represent $N'_{-1}[-1]$ and (b) in cycle 5 to represent $N'_0[-1]$. If the initial reciprocal approximation has a one-sided error, namely $e_0 \geq 0$, then $N'_0 < 2$ and $N'_0[-1] = 0$. As for $N'_{-1}[-1]$, the additional addend of the multiplier is not used in cycle 3. Hence, we could compute instead: $mul_{RZ}(N'_{-1}[0:28], F'_{-1}[0:14]) + N'_{-1}[-1] \cdot F'_{-1}[0:14]$.

Save a comparator in pipeline stage 0. One could compare $A < B$ during cycle 1 using the comparator in stage 3. This means that the comparator in stage 0 is not needed. Note that a conflict is not created between successive division operations since cycle 1 of the later operation corresponds to cycle 7 of the previous operation. The comparator in stage 3 is used only during cycle 8 so it is free to perform the comparison $A < B$ for the later operation.

Extra time for reciprocal approximation. In cycle 1, very little processing of F'_{-1} is required. This means that, if one needs more time to compute the reciprocal approximation, one can extend this computation so that it also takes place during part of cycle 1.

Instant operand normalization. We chose to use a multiplier in which both the first operand and the product have a digit in position $[-1]$. This can be avoided by instantly normalizing the first operand before feeding it into the multiplier so that the leftmost digit is always non-zero (this requires a row of MUXes).

D.7 Implementation of the dewpoint computation

In this section we provide additional details on the implementation of the dewpoint computation in the dewpoint circuit. Figure D.2 depicts the complete details of the dewpoint computation and recoding.

The dewpoint is computed as the sum of

$$dewpoint[0:24] \leftarrow N_{\text{INJ}}[0:23] + C_{dew}(mode) + carry_{23}(N_{\text{INJ}}[24:28]).$$

The representation of $N_{\text{INJ}}[0:28]$ is given as a carry-save string, and $C_{dew}(mode)$ can be easily be generated from the input of the rounding mode as

$$C_{dew}(mode) \triangleq \begin{cases} 2^{-(p-1)} & \text{if } mode = \text{RZ} \\ 2^{-p} & \text{if } mode = \text{RNU} \\ 0 & \text{if } mode = \text{RI}. \end{cases}$$

Note that the representation of $C_{dew}(mode)$ has at most one non-zero bit position that resides either in bit position [24] or in bit position [23].

The value of $|dewpoint[0:24]|$ does not need to be fully compressed, but it is to be fed into the Booth radix-8 multiplier as the recoded operand. For the implementation we apply a partial compression technique from [35] that allows compression from carry-save to Booth radix-8 recoded format through: (i.) the addition of a transformation constant ($Constant = 2^{-3} + 2^{0} + \cdots + 2^{-21}$) and (ii.) the application of a row of 3-bit adders with inverted ***most significant sum bits*** (MSSBs).

Note that the same recoding circuitry can be used to recode the operand represented in non-redundant representation. We separated the circuitry for recoding only to make the explanation simpler.

cycle	1st pipeline stage	2nd pipeline stage	3rd pipeline stage

1	$F'_{-1}[0:14] \leftarrow \frac{1-e_0}{B}$
	$D'_{-1}[0:28] \leftarrow B[0:23]$
	recode(F'_{-1})
	prepare($3D'_{-1}$)		

2	$N'_{-1}[-1:28] \leftarrow A'[-1:23]$ prepare($3N'_{-1}$)	$D'_0[0:28] \leftarrow mul_{RI}(D'_{-1}[0:28], F'_{-1}[0:14])$	
3		$N'_0[-1:28] \leftarrow mul_{RZ}(N'_{-1}[-1:28], F'_{-1}[0:14])$	compress(D'_0)

4	$F'_0[0:26] \leftarrow 2 - 2^{-26} - D'_0[0:26]$
	recode($F'_0[0:26]$)
	prepare($3N'_0[-1:28]$)
			compress(N'_0)

5		$N_{\text{INJ}}[0:28] \leftarrow \quad mul_{RZ}(N'_0[-1:28], F'_0[0:26])$ $+\text{INJ}_{rnd}(mode)$	

6	$dewpoint[0:24] \leftarrow \quad N_{\text{INJ}}[0:23] + C_{dew}(mode)$
	$+ carry_{23}(N_{\text{INJ}}[24:28])$
	recode($dewpoint[0:24]$)
	prepare($3 \cdot B$)
			$RC_1 \leftarrow \lfloor compress(N_{\text{INJ}}[0:28]) \rfloor_{23}$

7		$\alpha[0:47] \leftarrow \quad mul_{RZ}(B[0:23], dewpoint[0:24])$ $+(2^{-21} - A'[22:23])$	
8			$RC_2 \leftarrow RC_1 + 2^{-23}$ compute z-sign($\alpha[22:47]$) select between RC_1 & RC_2.

Table D.2: Schedule of operations of our FP division implementation with IEEE rounding for single precision operands

cycle	op 1st stage	op 2nd stage	op 3rd stage

1	$F'_{-1}[0:14] \leftarrow \frac{1-e_0}{B}$
	$D'_{-1}[0:60] \leftarrow B$
	recode(F'_{-1})
	prepare($3D'_{-1}$)		

2	$N'_{-1}[-1:60] \leftarrow A'[-1:52]$ prepare($3N'_{-1}$)	$D'_0[0:60] \leftarrow mul_{RI}(D'_{-1}[0:60], F'_{-1}[0:14])$	

3	$F'_0[0:56] \leftarrow 2-(D'_0[0:56]+2\cdot 2^{-56})$
	recode(F'_0)
	prepare($3D'_0$)
		$N'_0[-1:60] \leftarrow mul_{RZ}(N'_{-1}[-1:60], F'_{-1}[0:14])$	compress(D'_0)

4	prepare($3N'_0$)	$D'_1[0:60] \leftarrow mul_{RI}(D'_0[0:60], F'_0[0:56])$	compress(N'_0)
5		$N'_1[0:60] \leftarrow mul_{RZ}(N'_0[-1:60], F'_0[0:56])$	compress(D'_1)

6	$F'_1[27:56] \leftarrow 2^{-26}-(D'_1[27:56]+2^{-56})$
	recode($F'_1[27:56]$)
	prepare($3N'_1$)
			compress(N'_1)

7		$N_{\text{INJ}}[0:60] \quad \leftarrow mul_{RZ}(N'_1[0:60], F'_1[27:56])$ $+N'_1[0:60]+\text{INJ}_{rnd}(mode)$	

8	$dewpoint[0:53] \leftarrow N_{\text{INJ}}[0:52]$
	$+carry_{52}(N''_2[53:60])+C_{dew}(mode)$
	recode($dewpoint[0:53]$)
	prepare($3\cdot B$)
			$RC_1 \leftarrow \lfloor compress(N_{\text{INJ}}[0:60])\rfloor_{52}$

9		$\alpha[0:105] \leftarrow \quad B[0:52]\cdot dewpoint[0:53]$ $+(4-A'[-1:52])$	
10			$RC_2 \leftarrow RC_1+2^{-52}$ compute z-sign($\alpha[0:105]$) select between RC_1 & RC_2.

Table D.3: Schedule of operations of the proposed FP division implementation with IEEE rounding for double precision operands

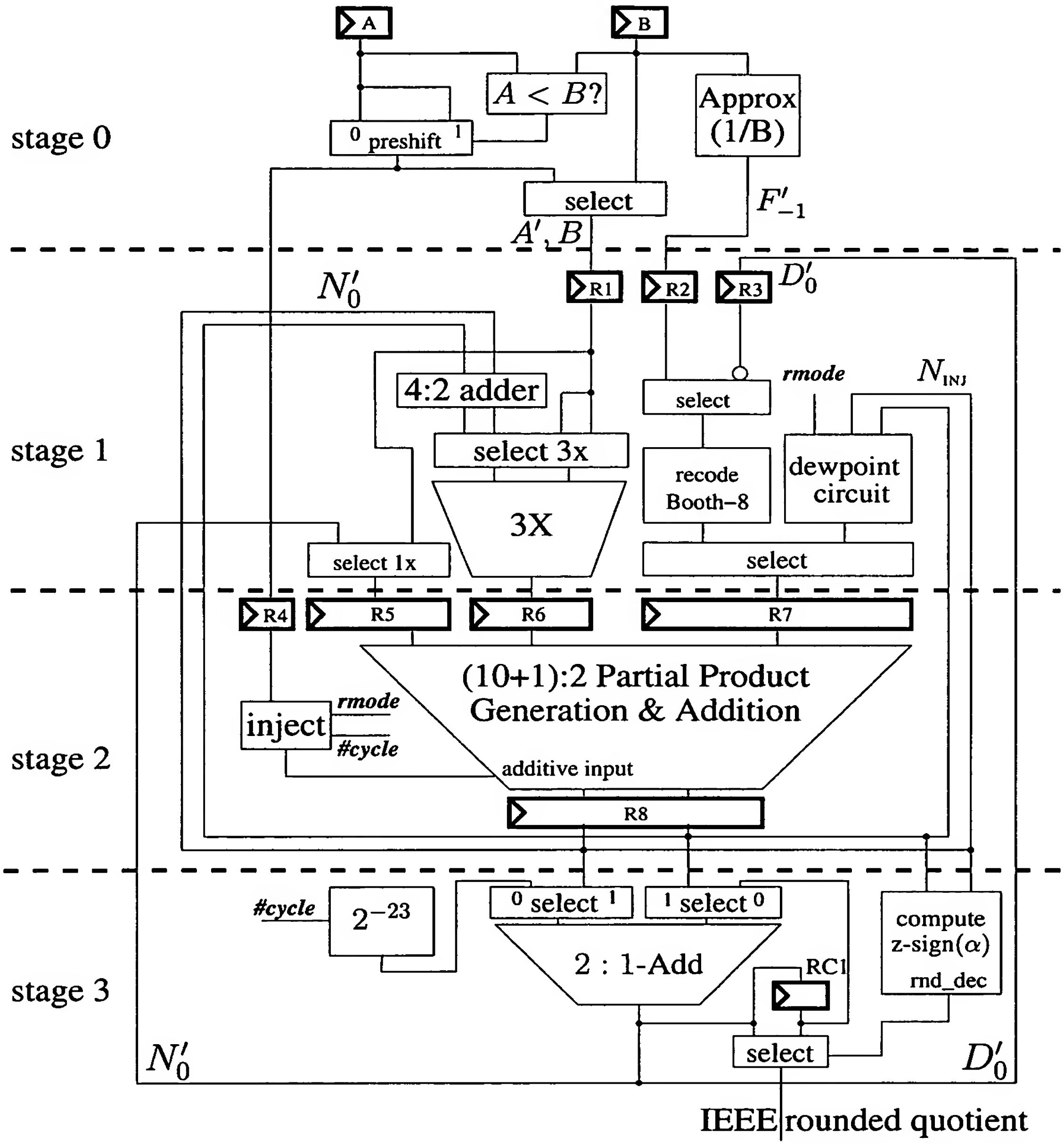

Figure D.1: Microarchitecture for single precision Division Implementation.

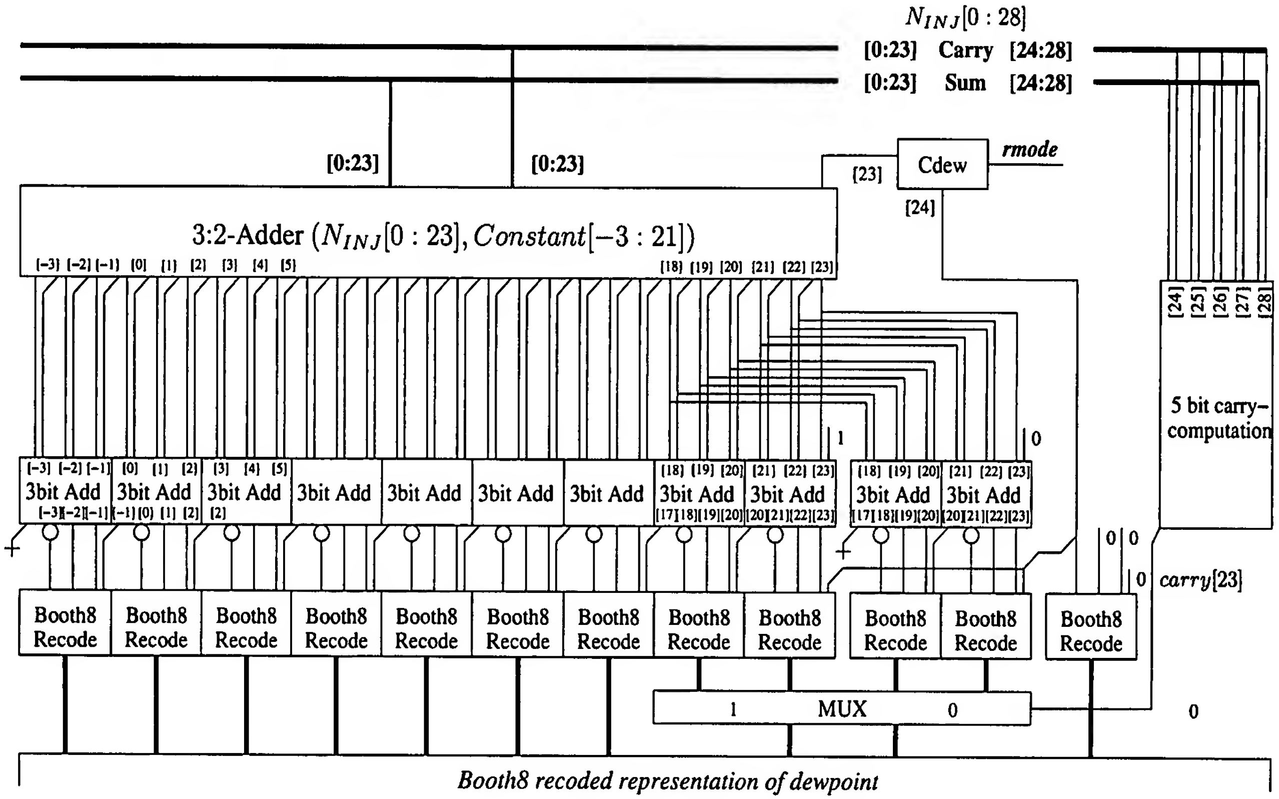

Figure D.2: Implementation of dewpoint circuit in pipeline stage 1.

D.8 Summary

We present a complete description of an implementation of Goldschmidt's algorithm for single precision using a microarchitecture that contains a 30×27 pipelined Booth radix-8 multiplier and an initial approximation with a relative error bounded by $2^{-12.71}$. This description includes detailed proofs of the error analysis, the rounding procedure, and various optimization techniques.

It is possible to extend the error analysis to double precision in a setting with a 57×62-rectangular multiplier and an initial approximation with precision $e_0 \leq 2^{-13.51}$. In Table B.3, the schedule of such an implementation is given. This double precision implementation achieves a latency of 11 clock cycles (including the initial approximation and IEEE rounding).